

SÉRIE
EDITORA Campus | SBC SOCIEDADE BRASILEIRA DE COMPUTAÇÃO

Bruno Feijó, Esteban Clua e Flávio S. Corrêa da Silva

Introdução à Ciência da Computação com Jogos

Aprendendo a programar com entretenimento

CAMPUS


CÓPIA NÃO AUTORIZADA É CRIME
ABDR
ASSOCIAÇÃO BRASILEIRA DE DIREITOS REPROGRÁFICOS
RESPEITE O DIREITO AUTORAL

Bruno Feijó, Esteban Clua e Flavio S. Corrêa da Silva

Introdução à Ciência da Computação com Jogos

Aprendendo a programar com entretenimento







Copidesque: Adriana Kraner
Revisão: Marília Pinto de Oliveira
Editoração Eletrônica: SBNIGRI Artes e Textos Ltda.

Elsevier Editora Ltda.
Conhecimento sem Fronteiras
Rua Sete de Setembro, 111 – 16º andar
20050-006 – Centro – Rio de Janeiro – RJ – Brasil

Rua Quintana, 753 – 8º andar
04569-011 – Brooklin – São Paulo – SP – Brasil

Serviço de Atendimento ao Cliente
0800-0265340
sac@elsevier.com.br

ISBN 978-85-352-3419-0

Nota: Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação ao nosso Serviço de Atendimento ao Cliente, para que possamos esclarecer ou encaminhar a questão.
Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso desta publicação.

CIP-Brasil. Catalogação-na-fonte.
Sindicato Nacional dos Editores de Livros, RJ

F328i Feijó, Bruno
Introdução à ciência da computação com jogos : aprendendo a programar com entretenimento / Bruno Feijó, Esteban Clua e Flavio S. Corrêa da Silva. – Rio de Janeiro: Elsevier, 2010.

Apêndice
ISBN 978-85-352-3419-0

1. Programação (Computadores). 2. Jogos por computador. 3. Animação por computador. 4. Computação. I. Clua, Esteban. II. Silva, Flavio S. Corrêa da. III. Título.

09-4873. CDD: 005.1
CDU: 004.42

Agradecimentos

Muitas pessoas contribuíram para a realização desta obra. Em primeiro lugar estão as nossas famílias, que se privaram do nosso convívio e da nossa atenção durante a nossa contínua e exagerada jornada extra de trabalho. Para eles vão os agradecimentos, literalmente infinitos, pelos apoio, compreensão e paciência. Bruno agradece à sua esposa Carmem e aos seus filhos Marcelo e Leonardo. Esteban agradece à sua esposa Luciana. Flávio agradece à sua esposa Renata e à sua filha Maria Clara.

Este livro não teria surgido sem o espírito inovador de Karin Breitman (professora e pesquisadora do Departamento de Informática da PUC-Rio) que, na qualidade de Diretora de Publicações da Sociedade Brasileira de Computação, deu toda a atenção e todo o estímulo aos autores.

Agradecimentos efusivos vão para a equipe da Editora Campus Elsevier que com rigor, competência e muita paciência nos acompanharam passo a passo. Em especial, gostaríamos de citar André Gerhard Wolff, Vanessa Vilas Bôas Huguenin e Silvia Barbosa Lima.

Sem a ajuda de Lucas Machado e Rodrigo Martins, o motor javaPlay não teria sido possível. Não temos como agradecer suficientemente a dedicação desses dois pesquisadores do VisionLab/PUC-Rio. Raros profissionais no país têm a competência desses dois pioneiros da área de jogos digitais.

A equipe por detrás deste livro é ainda maior e a muitos devemos palavras de agradecimento: Luciana Rocha Mariz Clua, por sua ajuda em design gráfico; Fernando Ribeiro e Pablo Bioni, pela arte nos elementos visuais dos jogos; Anselmo Antunes Montenegro e Dante Corbucci Filho, professores do Instituto de Computação/UFF, como consultores em Java; e Marcelo Pana-

ro Zamith, pela ajuda no material de demonstração colocado no site. Com o receio de não sermos suficientemente abrangentes e considerando o grande número de pessoas, nos limitamos a agradecer, em bloco, a todos os nossos alunos – nossa primeira fonte de inspiração e de colaboração na pesquisa e no ensino – por seus permanentes questionamentos, suas incuráveis mentes criativas e suas alegrias em aprender e ajudar.

Os Autores

Bruno Feijó possui graduação em Engenharia Aeronáutica pelo ITA, mestrado em simulação dinâmica pela PUC-Rio e doutorado em CAD (Computer-Aided Design) pela University of London. Atualmente é professor associado do DI (Departamento de Informática) da PUC-Rio, no Grupo de Computação Gráfica e Entretenimento Digital, e Coordenador do VisionLab/PUC-Rio – Laboratório de Visualização, TV/Cinema Digital, Games e Produção de Conteúdo Digital. É pioneiro em 3 áreas de pesquisa em Informática no país: CAD, Animação e Jogos. Foi um dos criadores da Comissão Especial da SBC em Jogos e Entretenimento Digital e foi o seu primeiro Presidente. Ele também é um dos fundadores do SBGames (Simpósio Brasileiro de Jogos e Entretenimento Digital). Foi o idealizador da Rede Brasileira de Visualização (RBV) ligada ao MCT. Também foi um dos principais colaboradores na idealização do curso profissionalizante do Núcleo de Pesquisa em Educação, criado pelo Instituto Oi Futuro. Na pós-graduação do Departamento de Informática da PUC-Rio, da qual já foi Coordenador, ministra cursos de jogos, animação e efeitos especiais. Na graduação do DI/PUC-Rio, da qual já foi Coordenador, tem ministrado os cursos de introdução à ciência da computação por cerca de 15 anos.

Esteban Clua possui graduação em Computação pela USP, mestrado pela PUC-Rio e doutorado também pela PUC-Rio. Atualmente é professor do Instituto de Computação da UFF e coordenador geral do UFF Media Lab, além de atuar na pós-graduação dessa universidade. Atua especialmente na área de Computação Gráfica em tempo real, Games, Realidade Virtual, GPUs,

visualização e simulação. É membro do conselho administrativo da ABRAGAMES, membro da Comissão Especial de Jogos e Entretenimento Digital da SBC e um dos pioneiros na área de pesquisa científica em jogos e entretenimento digital. Também é um dos criadores do SBGames e um grande entusiasta da área de jogos digitais.

Flávio S. Correa da Silva possui graduação em Engenharia de Produção pela USP, mestrado pela Politécnica da USP e doutorado em Inteligência Artificial pela University of Edinburgh. Atualmente é professor associado da USP, revisor do Journal of Information Technology Research e membro do corpo editorial da Applied Intelligence (Boston).

Prefácio

Estamos vivendo a revolução do mundo digital, onde ter noções de programação é tão essencial quanto saber o básico da matemática, da física, da economia, e quem sabe até dos primeiros socorros médicos... Até mesmo artistas e designers já perceberam que programar está na essência das novas mídias. Se você abriu este livro, muito provavelmente você ou é um fissurado por jogos, ou é um estudante, ou é um educador, ou pelo menos é um curioso incorrigível (uma das maiores qualidades do novo profissional criativo). Em qualquer desses casos, você deve estar buscando uma introdução à arte de programar que esteja mais próxima das novas demandas do mundo digital e que seja menos maçante para acompanhar. Você também pode estar procurando uma introdução à arte de desenvolver jogos digitais, quer porque você almeja ser um profissional da área, quer porque lhe disseram que o jogo digital é uma aplicação onde convergem os maiores desafios da computação (um jogo tem que ser multimídia, visualmente elaborado, distribuído, artificialmente inteligente ... e tudo isso rodando em tempo real). Como educador, muito provavelmente você deve estar buscando um texto mais leve e estimulante para a geração de jovens Y. Este livro pretende atender a todas essas expectativas.

Muita coisa mudou desde que a primeira linguagem de programação de alto nível foi lançada há mais de 50 anos. Desde então, as linguagens e os paradigmas de programação têm evoluído constantemente. Linguagens caem em desuso e novas surgem. E qual a melhor linguagem para aprender no momento? Essa é uma pergunta que vem sendo formulada desde o momento em que surgiram duas linguagens diferentes e os primeiros cursos de computação; e provavelmente continuará a ser feita até o dia em que não precisaremos mais dos computadores... Muito dificilmente haverá um consenso na resposta a essa

pergunta, mesmo entre os mais conceituados cientistas de computação do país. Não existe a situação ideal de que podemos aprender os conceitos da ciência da computação sem nos incomodarmos com as idiossincrasias e as complexidades de uma particular linguagem. Talvez o Lisp seja a que mais tenha se aproximado deste ideal (sua qualidade é atestada pelo fato de ser a segunda mais antiga linguagem do planeta ainda viva). Mas o Lisp está longe de ser uma opção para as novas demandas do mundo digital. A linguagem C já está distante dos novos paradigmas de programação. C++ é muito eficiente, especialmente no mundo dos jogos 3D, mas além de estar se tornando antiga, não é uma linguagem de fácil aprendizado. Neste livro optamos pela linguagem Java por várias razões. Primeiro, o Java possibilita aplicações em uma grande variedade de plataformas: internet nos servidores, PCs, celulares e sistemas embarcados. Segundo, o Java roda em mais de um sistema operacional (Windows, Linux, ...) e conta com um bom ambiente de desenvolvimento gratuito (NetBeans). Terceiro, o Java é uma linguagem orientada a objetos – um paradigma atual de programação. Quarto, Java é o ponto de partida para a programação da internet – a invenção que determinará o futuro da computação. Quinto, a próxima geração de linguagens orientadas a objetos será a evolução natural do Java (como já o é a linguagem C#). Sexto, Java serve muito bem para desenvolver jogos 2D – o tipo de aplicação que estamos usando neste livro.

Independente da escolha do Java, este livro procura, antes de mais nada, apresentar os conceitos e os bons hábitos da arte de programar. Dessa maneira, o livro está organizado em duas partes. A primeira parte se concentra na programação procedimental estruturada (capítulos 2 a 6) e a segunda parte é dedicada aos paradigmas de orientação orientada a objetos e de programação guiada a eventos (capítulos 7 a 9). Introduzir o conceito de orientação a objetos não é uma tarefa fácil, mas é indispensável no cenário atual da ciência da computação. Este livro procura ir trazendo esse novo conceito aos poucos, sem pretender esgotar o assunto, nem aspirar ser usado como um livro de referência. Este é um livro didático que funciona como ponto de partida para os modernos conceitos em ciência da computação. O projeto deste livro inclui uma terceira parte, disponível na internet, que pode sustentar um curso mais avançado e/ou pode dar continuidade a uma especialização na área de desenvolvimento de jogos:

www.inf.puc-rio.br/livroicc

A melhor maneira de usar este livro é implementar cada exemplo citado no texto e fazer os exercícios no final de cada capítulo. Um motor de jogo 2D, chamado javaPlay, foi especialmente desenvolvido para este livro. O javaPlay é conciso, didático, mas tem uma estrutura profissional, moderna e com possibilidade de ser usado com eficiência pela indústria. O javaPlay é apresentado em várias versões de graus crescentes de complexidade: Mini (a versão minimalista, para começar), Versão 00 (a versão intermediária) e Versão 0 (a versão-base mais completa). A Versão 1 deverá ser a que o leitor mais dedicado certamente criará.

Os apêndices complementam o estudo e servem de referência. Recomendamos, em especial, que sejam feitos os projetos de jogos do Apêndice B. Os amantes de desenvolvimento de jogos acharão muito interessante o Apêndice A. Os Apêndices C e D servem de referência para o motor de jogo javaPlay.

Este é um livro texto para um primeiro curso de introdução à ciência da computação, usando a criação de jogos digitais como instrumento de aprendizado e motivação. Esta obra é organizada em uma sequência de tópicos que correspondem a um curso de 64 horas, em blocos de 4h/semana (16 semanas, ao total). Sugestões de sequência de assuntos e material de suporte ao professor e ao aluno podem ser encontrados na página Web citada anteriormente. Uma possível sequência para 16 semanas pode ser:

semana 1:	apresentação e cap.1
semana 2:	cap.2
semana 3:	cap.3
semanas 4 a 6:	cap.4
semana 7:	cap.5
semana 8:	cap.6
semanas 9 a 11:	cap.7
semanas 12 a 14:	cap.8
semanas 15 e 16:	cap.9

Uma outra sequência de aprendizado pode ser proposta através de dois cursos, com mais tempo para elaborar os conceitos:

Curso I – Programação Procedimental Estruturada

Cap. 1 ao Cap. 5

Cap.6 como preparação ao Curso II

Curso II – Programação Orientada a Objetos e Guiada a Eventos

Cap. 7 ao Cap. 9

Material extra no site sobre estruturas de dados

A presente obra é resultado de anos de experiência dos autores no ensino de programação e de disciplinas de jogos, em três das melhores universidades do país. Este livro representa uma contribuição tanto para iniciar as pessoas na arte da programação como para ajudar na formação básica da mão de obra necessária para as oportunidades que surgem com a indústria do entretenimento (jogos e TV digital interativa) e com as aplicações de simulação em tempo real.

Sumário

APÊNDICES

Capítulo

1

Computadores

Harold Abelson e Gerald Sussman, no livro *Estrutura e Interpretação de Programas de Computador* (1996, MIT Press), apresentam uma dedicatória perfeita que resume o conceito de programação: "este livro é dedicado, em respeito e admiração, ao espírito que vive nos computadores". Os programas (ou seja, o software) são esses espíritos que habitam as máquinas, o hardware. Programar não é lidar com hardware, é lidar com processos, raciocínio, métodos e estratégias envolvendo artefatos abstratos. O presente livro é uma introdução a essa arte, que independe do conhecimento dos detalhes da máquina que a materializará. Para programar, você não precisa (nem deve) se preocupar com os detalhes de funcionamento do computador. Entretanto, precisamos entender a funcionalidade mais externa de seus principais componentes. Neste primeiro capítulo falaremos sobre essas máquinas maravilhosas, de uma maneira divertida, porém precisa (como é usual no mundo dos videogames).

1.1. A Long Long Time ago...

Os computadores fazem parte do nosso dia a dia, quase da mesma forma que o feijão com arroz, nossas tias ou as festas de aniversário. Antigamente existia feijão com arroz, tias e aniversários, mas o computador não... Ele foi inventado mais ou menos em 1943. (O termo "mais ou menos" se deve ao fato

de que um pouco antes desta data alguns cientistas já vinham desenvolvendo máquinas que poderiam ser talvez chamadas de computadores...)

Os primeiros computadores eram considerados simplesmente máquinas de calcular e fazer contas (Figura 1.1). Na verdade, foi para isto que os primeiros computadores foram criados: fazer contas de forma muito rápida. Não porque as pessoas tivessem muita pressa em obter os resultados, mas porque, fazendo contas rapidamente, elas poderiam começar a resolver problemas que exigiam muitíssimos cálculos. Um exemplo disso é a previsão meteorológica: uma boa previsão, para um país do tamanho do Brasil, requer aproximadamente 155.520.000.000.000.000 operações matemáticas. Mas não é preciso ser muito sofisticado para encontrar um bom exemplo: ao jogar um videogame de última geração podemos chegar a disparar aproximadamente 1.000.000.000.000 cálculos matemáticos.



Figura. 1.1 Foto do primeiro computador do mundo: o ENIAC. Essa máquina das cavernas media 17m x 9m, era capaz de fazer 38 divisões e 357 multiplicações por segundo e levava duas horas para calcular o mesmo que 100 engenheiros levariam em um ano apenas com lápis e papel. Seriam necessárias um milhão dessas máquinas para poder competir com um único PlayStation 3...

Pois bem, se você for uma pessoa prodígio e conseguir fazer 100 contas por segundo, levará mais ou menos 3 bilhões de anos (mais ou menos a idade da Terra...) para calcular a previsão do tempo para o dia de amanhã no Brasil.

O problema é que daqui a 3 bilhões de anos ninguém vai estar interessado em saber como será o dia amanhã, até porque além de esse dia já ter passado há muitíssimo tempo, talvez a humanidade nem exista mais ou quem sabe a raça humana tenha sido dominada pelos macacos.

Hoje em dia os computadores continuam sendo máquinas de fazer contas. E por incrível que pareça, um processador de última geração sabe fazer pouquíssimas coisas: realizar *operações fundamentais* (somar, subtrair, multiplicar e dividir) e comparar números (perguntamos a ele se 3 é maior que 2 e ele vai dizer: NÃO!).

Entretanto, os computadores que temos em casa fazem coisas muito mais sofisticadas do que as operações fundamentais: tocam MP3, executam jogos fantásticos e permitem entrar no Orkut. Pois bem: todas as coisas que o computador irá fazer devem estar sustentadas pelas operações básicas e devem ser ensinadas pelo homem. É aqui onde tudo começa...



NOTA

Talvez você já tenha ouvido falar da arquitetura RISC: *Reduced Instruction Set Computer*. Essa arquitetura está em contraponto com a tecnologia CISC: *Complex Instruction Set Computer*. Enquanto o RISC preza por um conjunto bastante reduzido de operações fundamentais, o CISC implementa um conjunto maior de instruções. O RISC predominou nas arquiteturas de supercomputadores, pelo fato de que a velocidade de execução de cada instrução é muito mais rápida e que processadores RISC usam menos transistores, sendo, portanto, mais baratos para projetar e construir.

1.2. al-Khowarizm

Imagine um programa de computador como um bolo. Para prepará-lo, precisaremos dos ingredientes, que, na nossa alegoria, correspondem às operações fundamentais. É preciso também ter a receita, que nos dirá como juntar cada elemento: quanto de farinha devemos usar, em que momento colocar no forno, quanto tempo assar etc. Em computação, chamamos a receita do bolo de *algoritmo*. Assim, para realizar operações mais complexas, devemos escrever uma receita, dizendo passo a passo o que deve ser feito, usando para tanto as operações fundamentais.



NOTA

O termo algoritmo não foi inventado pela computação... Conta-se que essa palavra surgiu devido a um persa matemático do século IX, chamado Abu Ja'far Mohammed ibn Mûsa al-Khowarizm (tente pronunciar este último nome e verá que se a história não for verdadeira, pelo menos tem um fundo de razão...). Esse matemático escreveu um livro chamado "*Kitab al jabr w'al-muqabala*", no qual ensinava como resolver problemas de matemática usando a ideia da receita de bolo, ou seja, algoritmos.

Um dos algoritmos mais antigos é o de Euclides, que foi escrito na Grécia em torno de 300 a.C. Esse algoritmo se propõe a encontrar o maior divisor inteiro comum entre um par de números. Para facilitar o entendimento, vamos escolher dois números: 3654 e 1365. O algoritmo diz: divida um dos números pelo outro e pegue o resto dessa divisão (... perceba o aspecto de receita de bolo que um algoritmo pode ter). Usando os números anteriores, ao dividir 3654 por 1365 teremos que o resultado é 2 e o resto é 924. O algoritmo continua, dizendo: pegue o segundo número da divisão (no caso, 1365) e coloque-o no lugar do primeiro, dividindo-o pelo resto obtido no cálculo anterior. Assim, passaremos a dividir 1365 por 924, que nos dará 1 como resultado e 441 como resto. O algoritmo diz para continuarmos esse procedimento até que o resto da divisão seja zero. O último resto obtido será o maior divisor comum. No exemplo escolhido teremos a seguinte sequência:

3654 ÷ 1365 → resto 924

1365 ÷ 924 → resto 441

924 ÷ 441 → resto 42

441 ÷ 42 → resto 21

42 ÷ 21 → resto 0

Dessa forma, obedecendo a receita de bolo, teremos que 21 será o maior divisor comum entre 3654 e 1365. Qual a lógica desse algoritmo? Certamente um matemático poderá provar por que ele funciona, mas independentemente da explicação do matemático, há sempre um tom de magia por trás dos algoritmos: seguindo cegamente uma série de regras e passos obtemos um resultado interessante e, em alguns casos, bem mais sofisticado e complexo do que o maior divisor comum entre dois números. Por exemplo, imagine um algoritmo que dite como um personagem de um videogame irá se comportar dentro de um mundo virtual, sem a interferência de um cérebro humano...

(esse tipo de problema está dentro de uma área da computação chamada Inteligência Artificial).

Uma vez desenvolvido um algoritmo, este pode ser guardado numa biblioteca, de forma a poder ser reutilizado quando queiramos. Mais ainda: existem algoritmos mais complexos, que podem vir a requerer o uso destes. Através de métodos de abstração, poderemos usá-los como se fossem mais uma operação fundamental.

1.3. Para que serve a computação?

Da mesma forma que se pode dizer que a tarefa do médico está em encontrar soluções para os nossos problemas de saúde, e a do cozinheiro, escrever e seguir as receitas que agradem o nosso paladar, o problema central da computação consiste em encontrar algoritmos que solucionem diversos problemas da melhor forma possível para as mais diversas situações do nosso dia a dia. Tal como a Medicina possui ramificações (cardiologista cuida do coração, neurologista do sistema nervoso etc.), a ciência da computação possui diversas ramificações:

- Banco de Dados: desenvolve algoritmos para gerar e manipular um volume grande de dados.
- Computação Gráfica: desenvolve soluções para sintetizar imagens numa tela plana, buscando simular (através de algoritmos) a iluminação, sombras, reflexos etc.
- Inteligência Artificial: busca algoritmos que de alguma forma procuram resolver problemas de planejamento ou de simulação de comportamentos.
- Otimização: procura algoritmos que sejam mais eficientes dos que os conhecidos (normalmente essa eficiência é medida em tempo).
- Computação Paralela: procura desenvolver algoritmos que são capazes de ser executados em vários computadores ao mesmo tempo.
- Entretenimento: ... faz algoritmos para dar razão de ser ao Playstation, ao Nintendo e ao Xbox!

Além dessas existem muitas outras áreas, tais como Redes, Engenharia de Software, Sistemas Distribuídos, Sistemas Operacionais, Compiladores etc. Além disso, constantemente, estão surgindo novas áreas, algumas com nomes

bastante curiosos, tais como Bioinformática, Computação Afetiva e Computação Quântica.

1.4. Qual é o idioma dos computadores?

Na relação homem-computador há um grande problema, semelhante ao de um ocidental que vai para a China: a comunicação. Além de falarmos numa língua distinta, nossa construção semântica é diferente do chinês. Não é suficiente traduzirmos palavra por palavra, pois as frases vão ficar estranhas e mal construídas .... Dentre as muitas diferenças, nós escrevemos sentenças da esquerda para direita e o chinês escreve da direita para a esquerda.

O computador possui uma linguagem bastante mais simples que a humana, mas ao mesmo tempo muito diferente. A linguagem do computador é fundamentalmente matemática. Além disso, a linguagem de máquina (todos os programas devem estar em linguagem de máquina) é essencialmente uma linguagem em notação binária: 1001110101010001001...

Enquanto nosso alfabeto possui 26 letras, o alfabeto da máquina possui apenas duas. Nossas palavras são compostas por combinações de letras, as palavras do computador são compostas por combinações de zeros e uns... (... não é uma linguagem que propicie muito a construção de poesias). É padrão da computação formar palavras compostas por 8 letras, que nesse caso chamamos de bits (cada bit é 0 ou 1). E as palavras são denominadas 1 **byte**.

Um processador que é chamado de 8 bits (como o Nintendo Entertainment System – NES ou Nintendinho, para os íntimos) consiste numa CPU que permite que uma palavra por vez seja processada. Um processador de 64 bits permite que 8 bytes sejam processados. O Playstation 2 chega a juntar duas palavras de 64 bits, criando blocos de 128 bits.

Como é lógico, a maioria dos seres humanos não sente a menor disposição ou interesse por escrever textos e algoritmos usando apenas zeros e uns. Aliás, se fosse assim haveria escassez de profissionais no mercado da informática, pois não há castigo pior do que escrever um texto em binário. Queremos, na medida do possível, usar letras e palavras da nossa linguagem natural.

É nesse contexto que entram as *linguagens de programação*. São linguagens definidas por pessoas, que de alguma forma se assemelham à nossa língua (ou pelo menos à dos seus criadores). Normalmente será através delas que faremos contato com as máquinas. Entretanto, como os computadores jamais

entenderão essas linguagens, existem programas especiais, chamados de **compiladores**, que também poderiam ser chamados de tradutores: esses programas convertem um texto escrito numa linguagem de programação específica para um arquivo em código de máquina, contendo apenas 0s e 1s.

Existem inúmeras linguagens distintas, inclusive algumas em fase de extinção e outras que já nem mais existem. Dizemos que uma linguagem é de baixo nível quando está mais próxima da linguagem da máquina e alto nível quando está mais próxima da humana (note nossa superioridade em relação às máquinas; o exterminador do futuro provavelmente não iria gostar disso...). Cada linguagem possui suas vantagens e desvantagens. Algumas foram desenvolvidas para fins específicos e são muito mais eficientes para um determinado tipo de aplicação. Outras foram desenvolvidas para fins genéricos.

Aprender a programar e a escrever algoritmos é algo que independe da linguagem que estamos usando, uma vez que há muitas semelhanças entre elas. Um dos objetivos deste livro é fazer você aprender como o computador "parece pensar" e como você pode resolver problemas através de um raciocínio algorítmico. Somente entendendo esse processo poderemos nos preocupar com o passo seguinte: fazer aplicações mais complexas, como jogos.

Algumas das linguagens mais conhecidas e usadas atualmente são as seguintes:

- C/C++: Há muito, muito tempo existia uma linguagem chamada BCPL, criada por Martin Richards. Em 1970, outra pessoa chamada Ken Thompson fez algumas melhorias nessa linguagem e chamou-a de "B". Em 1972, Dennis Ritchie e Ken Thompson fizeram novas melhorias e criaram a linguagem "C". Passados os anos, essa linguagem foi se tornando muito popular – até hoje é usada amplamente. Na década de 1980 surgiu um novo "paradigma" na computação: a programação orientada a objetos. Bjarne Stroustrup adaptou a linguagem C para esse novo conceito, dando origem ao C++, amplamente usado até hoje. A linguagem Pascal tem algumas semelhanças com o C. Entretanto, Pascal não chegou a ganhar muito espaço na indústria e acabou tornando-se uma linguagem mais acadêmica, por ser mais fácil e simples que o C.
- Java: Na década de 1990, James Gosling, Bill Joy e outros pesquisadores da Sun Microsystems começaram o desenvolvimento de uma linguagem chamada Oak. O objetivo deles era inicialmente uma linguagem simples que fosse capaz de controlar microprocessadores embarcados

(processadores inseridos dentro de diversos aparelhos, tais como *set-top boxes* para as futuras TVs interativas, videocassetes e torradeiras de pão computadorizadas, bem como para os primeiros PDAs, Personal Data Assistance – hoje mais conhecidos como Palms, e celulares). Para tanto, a linguagem projetada reuniu as seguintes características: independente de plataforma, compacta e segura. A TV interativa e os PDAs, principais plataformas motivadores do Oak, não avançaram como se pensava nos anos seguintes, mas por outro lado foi justamente nessa época que ocorreu o *boom* da Internet. Assim, a Sun rapidamente adequou a linguagem para esse mercado, passando a se chamar Java. Atualmente ela se caracteriza por ser uma linguagem de alto nível que usa o conceito de Máquina Virtual (*Virtual Java Machine*): um programa que recebe o código gerado pelo compilador e o executa, comando a comando, como se fosse uma CPU. Há algumas variações dessa linguagem, como por exemplo o Java MicroEdition (J2ME), customizado e simplificado para ser executado em celulares.

- C#: É uma linguagem moderna, desenvolvida pela Microsoft. Possui conceitos do C, C++ e Java.
- ASP e PHP: À medida que as aplicações para a Web progrediram, foram sendo necessárias linguagens mais adequadas e específicas para essa plataforma. Essas novas necessidades consistem em acesso a banco de dados remotos ou execução de programas que estão em computadores distantes. Para esse tipo de finalidade surgiram linguagens como o ASP (*Active Server Pages*) e a PHP.
- Lua: Há uma classe de linguagens especiais, denominadas linguagens de extensão ou linguagens de *script*, que não possuem a noção de um programa principal e funciona embarcada (*embedded*) em um programa anfitrião (chamado de hospedeiro). *Scripts* são distintos do código do hospedeiro. Em geral, essas linguagens são interpretadas, isto é: você pode executar o programa linha a linha, obtendo imediatamente a resposta de cada comando. Em geral, elas são projetadas para que não programadores sejam seus principais usuários, como ocorre nas seguintes linguagens especialmente projetadas para ferramentas proprietárias de modelagem e animação: MaxScript (linguagem script do 3DS Max), MEL (Maya Embedded language) e Flash Action Script (para o Adobe Flash Player). Lua é uma linguagem *script* de propó-

sito geral, escrita em C, desenvolvida pela PUC-Rio (www.lua.org), que está se tornando muito popular na indústria de videogames. Outras linguagens de script de propósito geral semelhantes ao Lua são o Phyton, o Pearl e o Javascript. O Phyton é particularmente popular na plataforma de celulares.

1.5. Insetos

Um dos maiores desgostos de um jogador de futebol é perder uma partida. E qual o maior desgosto de um profissional da computação? É quando ele encontra um inseto no seu programa... Se alguém usar esse termo numa reunião de analistas de sistemas, talvez as pessoas irão olhar com uma cara de “não entendi”, mas se for usada a palavra em inglês (*bug*), vão entender rapidamente do que você está falando: seu programa não está funcionado ou está apresentando resultados não esperados ou incorretos. Hoje esse termo é tão comum na computação, que outros termos muito usados derivam dele, como *debugar* (tirar um inseto do programa). Existem programas, denominados *debuggers* (o Buggzila é um exemplo), que se merecessem um ícone poderia ser o de um inseticida... A função destes programas é auxiliar o programador a encontrar onde estão as possíveis causas dos problemas.

Infelizmente os *bugs* estão sempre presentes na vida de um cientista da computação, por mais *hacker* que ele venha a ser. Aliás, quanto maiores e mais complexos os programas, piores são os insetos que podem vir a aparecer.

Os computadores são cegos e eles vão fazer as coisas exatamente conforme foram solicitados a fazer, levando exatamente ao pé da letra aquilo que foi dito a eles. Por exemplo, se um programador desatento inverter o sinal de uma variável que representa a aceleração da gravidade, o jogo pode tanto fazer pedras caírem para cima, como pode gerar um valor imprevisível para a variável que representa o número de vidas do jogador. Como *bug*, o jogo pode apresentar o sintoma de acabar de uma hora para outra, sem motivo, pois esse número imprevisível pode ter sido negativo, e a vida não pode ser negativa...

Aprender a formalizar e objetivar ao máximo todos os problemas e conceitos é uma das primeiras coisas que um aprendiz de computação deve fazer. Quando a ideia ou o algoritmo que se deseja programar estão bem formalizados, é mais fácil encontrar os *bugs* de um programa. É mais ou menos como uma pessoa que se perde numa parte da cidade que conhece bem.

O termo *bug* surgiu da seguinte forma: em 1945, Grace Murray Hopper estava trabalhando num dos primeiros computadores da história, o Mark II Aiken Relay Calculator, na Universidade de Harvard. Estavam executando um programa para cálculos matemáticos, e os resultados estavam muito estranhos. Após revisar minuciosamente dezenas de vezes o programa e não ter encontrado nenhum problema, um técnico resolveu verificar o hardware. Após uma rigorosa busca, viu que uma das válvulas nunca estava se fechando, pois havia uma pequena mariposa morta impedindo o contato (tinha sido eletrocutada...). Ao tirá-la, o programa passou a funcionar corretamente. No relatório de operações daquele dia, Grace Hopper colou a mariposa na folha (ver Figura 1.2), tornando-a, quem sabe, a mariposa mais famosa do mundo.

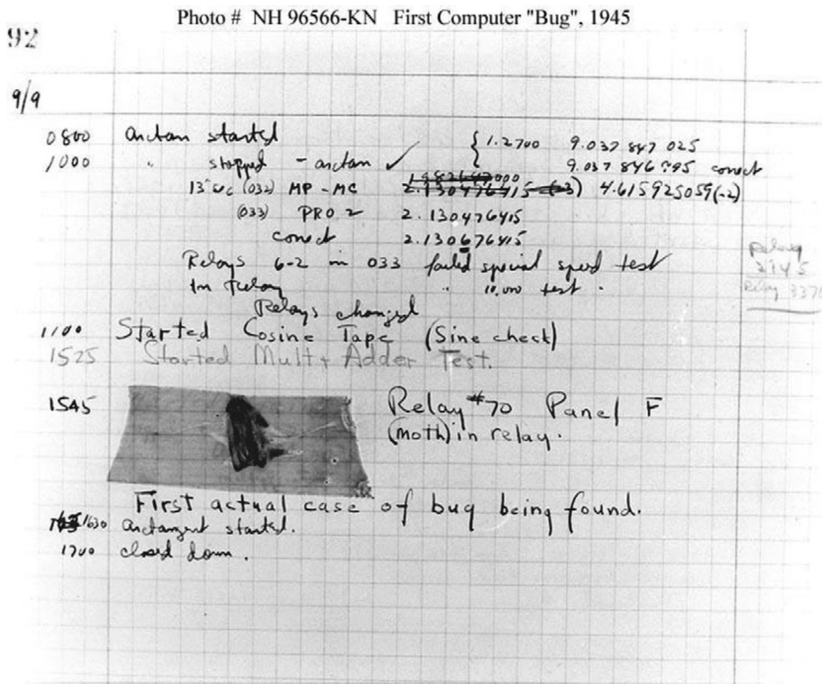


Figura 1.2 Relatório de operações do primeiro *Bug* da era da informática. Também é a foto do inseto mais famoso do mundo.

1.6. Sintaxe e semântica

Ao falar para uma pessoa: "Corra! Vindo ladrão está", ela irá olhar estranho e pensar (corretamente) que você não sabe português. Isso porque, nas regras gramáticas da nossa língua, é necessário seguir certa ordem para concatenar as palavras: sujeito seguido de verbo e os complementos. Quando

não se respeitam os padrões estabelecidos, ocorre um erro de sintaxe. Em programação também pode haver erros de sintaxe quando não forem obedecidas regras da linguagem. Em Java, por exemplo, as regras gramaticais dizem que para declarar uma variável deve-se seguir esta sintaxe:

```
tipo nomeDaVariavel;
```

tal como,

```
int velocidade;
```

Caso o programador escreva

```
velocidade int;
```

estará cometendo um erro de sintaxe. Entretanto, diferentemente do que ocorre na vida humana, os computadores não perdoam erros e ao compilar esse trecho de código será dada uma mensagem dizendo que houve um erro de sintaxe, não sendo possível prosseguir enquanto não for corrigido.

Os erros de sintaxe são fatais para um programa, porém são facilmente detectáveis. O segundo tipo de erro, mais grave que o anterior, consiste no erro de semântica. Se dissermos para alguém "o número PI vale 7.1415" e essa pessoa não sabe nada de trigonometria, pode ser que ela não desconfie que o autor da frase também não sabe trigonometria e disse uma coisa totalmente absurda. Esse caso é mais grave de ser detectado, pois requer uma análise de significado e nem sempre é possível detectarmos o erro. Os erros de semântica são erros conceituais, e dificilmente algum compilador poderá nos ajudar.

1.7. Arquitetura de computadores

Para pilotar um avião não é necessário que se conheça a fundo toda a arquitetura da aeronave (como funcionam as turbinas, processo de queima de combustível, princípios de aerodinâmica ou de onde veio a borracha do pneu do trem de pouso). No entanto, é muito interessante que um piloto tenha algum conhecimento sobre alguns desses tópicos.

De igual maneira, um programador não precisa conhecer a fundo como é o hardware de um computador para poder desenvolver softwares, mas é muito interessante que ele entenda um pouquinho.

A área denominada arquitetura de computadores é o lado "duro" da computação: preocupa-se por entender e estudar como as máquinas são sob o ponto de vista do hardware.

A Figura 1.3 representa um diagrama básico de uma arquitetura de um computador padrão.

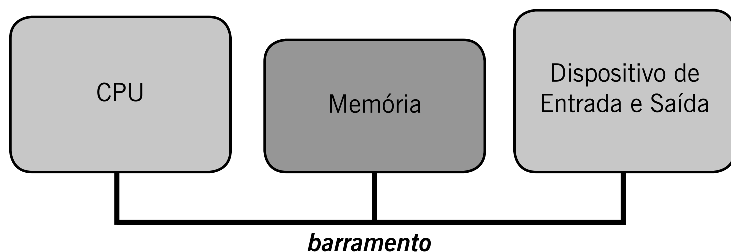


Figura 1.3 Arquitetura básica de um computador.

A CPU (*Central Processor Unit*) é o núcleo de qualquer arquitetura. Ela é a responsável pelo processamento em si, e todos os demais componentes têm a função de mandar dados para ela ou receber dados processados por ela. A memória é responsável por armazenar todos os dados, sendo alguns temporários e outros definitivos. Os dispositivos de entrada e saída são todos os elementos que levam informações para dentro do computador (teclado, leitor de DVD, joystick etc.) ou que mostram dados vindos da máquina (monitor, impressora, caixas de som etc.). O barramento é uma via por onde os dados trafegam de um componente para outro. Vejamos a seguir cada componente com mais detalhes.

CPU

A CPU é basicamente uma máquina de fazer operações aritméticas e lógicas. Ela também tem uma arquitetura interna, e se fossemos mostrar o diagrama real de uma CPU, talvez você decidisse abandonar a computação para sempre. Grosso modo, a arquitetura da CPU é apresentada na Figura 1.4.

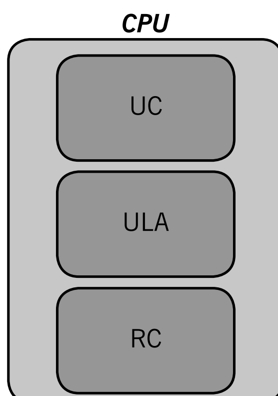


Figura 1.4 Arquitetura básica de uma CPU.

A Unidade de Controle (UC) é responsável por controlar os dados que estão chegando, saindo e sendo processados. Ela é uma espécie de gerente, que irá garantir que cada componente da CPU esteja recebendo o que deve receber e irá ajudar a despachar os dados para o mundo externo.

A Unidade Lógica-Aritmética (ULA) é responsável por efetuar as operações aritméticas (somas, multiplicações, divisões, subtrações etc.) e as operações lógicas (comparações e conexão de expressões lógicas).

Os Registradores (RC) são espaços onde os dados são colocados, para serem processados. Assim como nós precisamos escrever numa folha de papel dois operandos e posteriormente escrever o resultado de uma expressão aritmética, a CPU precisa escrever (nos registradores) os valores que serão usados numa determinada operação.

Assim como o nosso cérebro é basicamente formado por neurônios, a CPU é basicamente formada por transistores. Os primeiros computadores usavam na verdade válvulas, precursores dos transistores, e podiam medir até 3 a 4 cm. Com o passar do tempo, os transistores foram diminuindo muito de tamanho, sendo possível, num mesmo espaço físico, colocar mais e mais transistores, aumentando dessa forma o poder de processamento. Hoje em dia um transistor mede 45 nm (para se ter uma ideia, no espaço de uma válvula é possível colocar 4 milhões de transistores).

Devido a essa diminuição de tamanho dos transistores, recentemente surgiu uma nova tendência: começaram a colocar vários processadores dentro de uma mesma CPU. Assim, alguns problemas da computação podem ser divididos em pedaços, de forma que cada processador cuide de uma parte, o que permite resolver problemas de forma muito mais rápida.

Quando vamos pintar uma parede, precisamos de uma série de instrumentos: tintas, pincel, escada etc. Se deixarmos todos esses materiais distantes da área de trabalho, gastaremos um tempo grande tendo de nos deslocarmos de um canto a outro para ir pegando cada elemento. Assim, sempre que possível, deixamos tudo ao nosso alcance, de maneira a otimizar o tempo. Entretanto, nem sempre podemos deixar à mão, pois em geral podemos não ter espaço suficiente. Todos os dados que serão processados pela CPU residem em alguma memória (RAM, HD, pen-drive, DVD etc.). O processo de enviar um dado para a CPU é um pouco demorado, se comparado ao tempo que ela leva para fazer o cálculo propriamente dito. Essa demora se deve a vários fatores: esperar que o cabeçote do disco rígido se mova ao local onde está o dado,

colocar as informações no barramento de dados etc. Assim, a arquitetura de uma CPU apresenta também uma memória especial, denominada Cache: ela é muito rápida, pois está dentro da CPU. Uma série de dados que serão usados com frequência podem ser colocados nessa memória. Além disso, muitos cálculos são feitos usando resultados de cálculos preliminares. A CPU, nesse caso, pode armazenar os dados temporários na memória Cache, otimizando o tráfego de dados.

Memória

A memória consiste num espaço onde os dados são armazenados. Quanto maior o espaço dessa memória, maior o volume de dados que poderemos lidar; mas, em compensação, maior será o trabalho para encontrar as informações. Há inúmeros dispositivos de memória: disco rígido, memória flash, DVD, blu-ray, cartões de memória e muitos outros que ainda estão por ser inventados. Para um programa, a memória mais importante é a RAM: ela é volátil, e, sempre que desligamos o computador, todos os dados que estão nela se perdem. A vantagem dessa memória é que ela está diretamente ligada ao barramento, de forma que a CPU consegue ler e escrever nela de forma rápida e eficiente (vimos que a CPU prefere a memória cache, que está dentro dela mesmo, mas ela é muito pequena, se comparada à RAM). A memória RAM pode ser vista como uma lista de bytes, sendo que cada byte tem um endereço próprio. Quando o programa vai buscar algum dado, ele tem de informar qual é o endereço em que essa informação reside. Caso ele não saiba qual é o endereço, terá de fazer uma busca exaustiva, podendo travar o processo por um bom tempo até encontrar. Se enchermos toda a memória RAM de dados, a CPU poderá usar o disco rígido ou outro dispositivo do sistema. No entanto, o tempo de acesso será exorbitantemente maior, e o usuário da aplicação poderá ficar muito chateado... Mais adiante veremos como lidar com o problema de endereçamento de memória.

Barramento

Assim como a vida de uma cidade consiste em pessoas indo e vindo para diversos locais, a vida de um sistema computacional ou programa se resume a um vai e vem de dados. O usuário entra com dados, estes são processados e transformados em outros dados, para posteriormente serem armazenados ou

devolvidos ao usuário, que por sinal poderá enviar mais dados... O que seria de uma cidade se não houvesse ruas, avenidas e passarelas? Provavelmente teríamos de cavar túneis para ir de um lugar a outro, mas teríamos então um novo tipo de via de comunicação.

Os barramentos nada mais são que canais por onde os dados trafegam. Numa cidade, a largura mínima de uma rua deve ser a largura de um carro. Num computador a largura mínima deve ser a de um byte, já que os dados são em geral compostos por conjuntos de bytes. Assim, dizemos que um barramento de 8 bits permite que um byte seja transportado por vez de um canto a outro. O Nintendo Entertainment System (famoso nintendinho) possuía uma arquitetura de 8 bits. À medida que os jogos foram se tornando maiores e compostos por mais dados (mais texturas, mais modelos, mais código), o barramento de 8 bits começou a atrapalhar o processamento, tornando-se um gargalo para a aplicação (algo semelhante ao que acontece quando estamos numa estrada de 5 pistas e num determinado momento, devido a obras, ela vira uma única pista ... e estamos na hora do rush). Assim, novas arquiteturas foram surgindo, permitindo que mais dados trafegassem por vez: 16 bits (Mega Drive, Super Nintendo), 32 bits (Super Genesis), 64 bits (Nintendo 64, Playstation II) e 128 bits (Playstation 3, Xbox 360). Obviamente esse crescimento não ocorreu apenas nos games: processadores comuns hoje já trabalham com 64 ou 128 bits nos seus barramentos.

1.8. Dispositivos de entrada e saída

Os dispositivos de entrada e saída são a porta de comunicação do nosso mundo com o mundo do computador. Sem eles o computador seria de uma tremenda inutilidade, pois não seria capaz de receber ou mostrar nenhum tipo de dado e perderia todo o seu propósito.

Alguns dispositivos de entrada são: teclado, mouse, joystick, Web cam, câmera fotográfica, scanner, leitor de retina e sensor de presença. Como dispositivos de saída podemos citar os seguintes: monitor, impressora e caixa de som. Ao desenvolver programas teremos de criar mecanismos para acessar esses dispositivos de entrada e saída.

Capítulo

2

Variáveis, Tipos de Dados e Expressões

Ao aprendermos um idioma humano, ele pode não servir para nada quando tentarmos aprender um outro idioma (se você sabe finlandês, isso não ajuda muito para o seu aprendizado de árabe ou de uma das línguas faladas na ilha de Java). Em computação, embora haja um número enorme de linguagens diferentes, ao aprendermos a programar numa delas, de alguma forma estaremos nos capacitando para entender todas as demais. Isso ocorre porque há uma semelhança grande nos conceitos envolvidos. Os paradigmas que começarão a ser apresentados a partir deste capítulo serão válidos para qualquer outra linguagem que você venha a usar futuramente.

Há quem diga que programar é uma arte. Independentemente se isso é verdade ou não, podemos ter a certeza de uma coisa: assim como é impossível aprender a pintar apenas lendo um livro de arte, é impossível aprender a programar apenas lendo este livro que está em suas mãos! Portanto, a partir deste momento, recomendamos que você procure colocar em prática todos os conceitos que forem sendo apresentados, para obter o maior proveito possível deste seu primeiro livro de programação. A maioria dos códigos deste livro possui seus respectivos arquivos de projetos na página Web do livro, onde também estão instruções de como usar um ambiente de programação Java.

Já foi dito, mas voltamos a dizer: o computador é extremamente "burro". Não sabe fazer quase nada. Além disso, não podemos errar nem sequer uma

vírgula quando vamos ensiná-lo a fazer algo, pois ele obedece cegamente a tudo o que pedimos. Um programa não pode ter nenhuma ambiguidade e nenhuma incorreção ou imprecisão. Assim, um dos primeiros exercícios que o aprendiz de computação deve fazer é tentar se abstrair dos detalhes e descrever conceitualmente o problema que deseja resolver. Neste livro, teremos inúmeras oportunidades para exercitar essa habilidade, não se preocupe.

Uma forma comum de trabalhar com tal abstração consiste em escrever um programa no formato chamado de "**pseudocódigo**", isto é: escrevemos, em sentenças curtas e objetivas, os passos do algoritmo, sem preocupações com a sintaxe ou com os detalhes da linguagem.

Há uma tradição na computação que não pretendemos violar neste livro: o primeiro programa da sua vida deve ser o "Hello World" (e faz parte da tradição ser em inglês...). Esse programa apenas imprime na tela a mensagem Hello Word. O pseudocódigo para o "emocionante" programa está no Código 2.1. A concretização desse código simples em comandos Java é o assunto da próxima seção. O restante deste capítulo apresenta os mecanismos de entrada e saída de dados, as variáveis que armazenam valores, os tipos de dados que podem ser representados e as expressões que podem ser construídas com os elementos básicos da linguagem Java. Estes serão seus primeiros passos na arte de programar.

Código 2.1 Pseudocódigo para o Hello Word

```
   Iniciar programa
Imprimir "Hello Word"
Finalizar programa
```

2.1. Primeiros passos, ou melhor, primeiras palavras em Java

Para que você possa iniciar a prática do Java, compilando e executando programas, vejamos como seria o programa Hello Word em "javanês". Para tanto, rode o programa do Código 2.2.

Código 2.2 Hello Word

```
class HelloWorld
{
   public void main(String[] args)
   {
      System.out.print("Hello World!"); // imprime mensagem
   }
}
```

Ao comparar esse código com o pseudocódigo apresentado anteriormente, você pode perceber a diferença de abstração: no código em Java temos de levar em conta vários detalhes da linguagem, o que nos leva a algo um pouco mais rebuscado. Vamos procurar entender em detalhes o que o Código 2.2 está fazendo.

Java é uma linguagem orientada a objetos. Os objetos são originados de classes. Embora as classes e os objetos sejam de fundamental importância, a tal ponto que todos os jogos e grande parte dos programas sejam feitos usando esses conceitos, deixaremos para apresentar o tema em detalhes nos capítulos 7 a 9 deste livro. Por enquanto, é importante ressaltar que todo programa tem de ser composto por classes. Pode-se imaginar que cada classe é uma espécie de artefato ou peça de uma máquina. A máquina por si só também é um artefato, portanto, uma classe. No momento, basta sabermos que uma classe contém métodos e estes realizam tarefas. O método `main` é como qualquer outro método, exceto que ele é diretamente chamado pela máquina virtual Java. Às vezes, informalmente, chamamos os métodos de funções, mas para ficarmos dentro do espírito do Java vamos evitar usar o termo função que é mais adequado para outras linguagens (como a linguagem C).

Quando uma fábrica de automóveis monta um carro, basicamente o que ela faz é juntar várias peças. Algumas delas podem ter sido feitas na própria fábrica, mas algumas podem ter sido feitas em outros lugares. A fábrica não precisa se preocupar em saber como é a peça por dentro, basta saber o que ela faz e como encaixá-la no automóvel. Em programação orientada a objetos, isto também ocorre: estaremos lidando constantemente com peças (objetos e classes), algumas delas são fabricadas (programadas) por nós mesmos e outras são fornecidas por terceiros. Uma das grandes vantagens de orientação a objetos é que ao usar um objeto não é necessário se preocupar como é o código dele, mas apenas como usá-lo.

A primeira linha de um programa dá o nome a uma classe, que basicamente será o nosso automóvel, ou seja, a peça principal. Dentro dessa classe deve estar o nosso programa. A sintaxe para criar um programa simples segue o padrão do Código 2.3.

Código 2.3 Padrão para programas simples em Java

```
class nome_do_Programa
{
   // espaço para declaração de variáveis e funções
   public void main(...) // método principal, que nem todas as
   {                     // classes precisam ter
   }
}
```

Os símbolos `{` e `}` são de extrema importância e usados em praticamente qualquer linguagem, mesmo as que não são orientadas a objetos. Eles definem um **bloco** de um código e correspondem mais ou menos a dizer "Aqui começa" e "Aqui termina". Um programa pode conter blocos dentro de blocos, como pode ser visto no Código 2.4 (escrito em pseudocódigo). Os outros símbolos usados no Código 2.2 são os seguintes: o símbolo `;` que indica o fim de um comando e o símbolo `//` após o qual tudo é considerado comentário até o final da linha.

Código 2.4 Exemplo de blocos (em pseudocódigo)

```
1   class jogo
2   {
3      realize leitura de teclado
4      {
5         ...  // Aqui chamamos mecanismos de ler teclado
6      }
7      Movimente o personagem
8      {
9         ... // Usando a entrada, movimentamos o personagem
10        ... // conforme desejado
11     }
12  }
```

No Código 2.4, dentro do bloco delimitado pelas linhas 4 e 6 deverá haver uma série de códigos encarregados da leitura de dados do teclado. Apenas quando essas tarefas acabarem de ser realizadas terá início o bloco seguinte (linhas 8 a 11) correspondente a movimentar o personagem, já usando os dados de entrada do teclado do escopo anterior.

A linha 3 do Código 2.2 é de extrema importância: ela dita que este é o método principal de toda a classe `HelloWord`. Como o exemplo é muito simples, esse é o único método em todo o código. Entretanto, mais adiante, vamos ver que dentro de uma classe e de um programa pode haver muitos métodos. O programa sempre começa executando o método principal, ou seja, o `main()`. Pode haver classes que não contêm o método `main()`. Nesse caso, elas terão de ser usadas ou disparadas por outra classe que contenha o `main()` ou por classes que não contêm o método `main()` mas que são chamadas por outra que contém o `main()`, ...ou, bom, você já sabe: uma mensagem de erro aparecerá na tela do seu computador. A palavra reservada `public` diz que esse método pode ser chamado por qualquer um que esteja fora da classe a qual pertence. Em geral, o método `main()` deve sempre ser `public`, pois deve ser disparado pelo inter-

pretador Java que, de certa forma, é externo à classe em questão. Na próxima seção vamos ver o que significa o método `System.out.print`.

2.2. Saída de dados

Enquanto não houver uma batalha entre máquinas e homens e elas ganharem, os programas e os computadores continuarão sendo tanto mais úteis quanto melhor se comunicarem com os humanos. Assim, todo sistema precisa ler dados e enviá-los ao mundo externo. A leitura de dados pode ser feita através de um teclado, um mouse, um joystick, uma webcam ou, no futuro, até mesmo através de eletrodos colados na sua cabeça. A escrita para o mundo externo pode ser feita no monitor, impressora ou caixa de som, dentre outras tantas possibilidades. O monitor (*display*) é o dispositivo mais usual de saída; portanto, caso não digamos nada, os dados serão muito provavelmente enviados para lá. É por isso que costumamos chamá-lo de dispositivo padrão do programa. Vale lembrar que hoje podemos ter muitos programas sendo executados ao mesmo tempo (você pode, simultaneamente, estar escrevendo um texto num editor, ouvindo mp3 e até falando com alguém no chat). Assim, cada programa em geral criará um terminal de saída, que, num primeiro momento, podemos imaginar como sendo uma janela (mas atenção: apenas num primeiro momento...).

Imprimir uma mensagem na tela não é uma tarefa simples e trivial como parece. O sistema deverá ler os caracteres, enviá-los para a saída de vídeo, ver em que lugar da tela serão impressos, converter os caracteres para um sistema hexadecimal etc. Entretanto, toda essa burocracia está encapsulada por algo que chamamos de método, que corresponde a um bloco de programa, com relação ao qual não precisamos nos preocupar como funciona por dentro, mas apenas saber como usar. As linguagens oferecem muitos métodos prontos. O programador poderá também desenvolver seus próprios métodos, possibilitando que ele seja usado em diversos trechos do programa ou até mesmo em outros programas que serão feitos futuramente.

Os métodos consistem numa importante estratégia de encapsulamento de código, isto é: escrevemos um programa para resolver um assunto específico e posteriormente não precisaremos nos preocupar mais com esse trecho de programa; apenas reutilizaremos o que já foi escrito, invocando o método

criado. Dessa forma, precisaremos apenas nos preocupar em como usar o método:

a. Saber qual é a sintaxe do método.
b. Saber o que o método faz ou resolve.
c. Saber quais são os parâmetros ou argumentos do método.

Para escrever dados no mundo externo, usamos o seguinte método:

```
System.out.print(“texto a ser impresso”)
```

Esse é um nome de método composto, uma vez que o método print pertence a um objeto chamado out, que, por sua vez, pertence a um objeto chamado System. Usamos o **operador ponto** (***dot operator***) para acessar objetos e métodos. Seguindo a estratégia de uso proposta anteriormente, temos que:

a. A sintaxe do método é System.out.print([conteúdo a ser impresso])
b. Esse método imprime algo na tela. O cursor permanece exatamente ao final do último caractere impresso; de maneira que, se a função for novamente chamada, a impressão continuará na sequência.
c. O único parâmetro é o conteúdo que desejamos imprimir.

O método “irmão” System.out.println(<*conteúdo a ser impresso*>) é muito parecido; porém, ao término da impressão, o cursor irá realizar automaticamente uma quebra de linha. Assim, ao chamar novamente uma impressão, o conteúdo irá aparecer na linha seguinte.

Para entrar com dados, a tarefa é um pouco mais complicada. Para esse processo será necessário usar o conceito de variáveis. Portanto, deixaremos o assunto para mais adiante (Seção 2.5).

2.3. Comentários

Na linha 5 do Código 2.2 e nas linhas 5 e 9 no pseudocódigo do Código 2.4, aparecem explicações, no bom e velho português, dizendo o que estamos pretendendo fazer naquele trecho de programa. Chamamos a essa estratégia de “comentários”. Em muitas situações é importante explicarmos em linguagem natural o que significa um trecho de código. Obviamente essa explicação não é para o computador, mas para aqueles que lerão o programa (um colega de trabalho, um cliente ou até mesmo o professor corrigindo uma prova...). Para inserir um comentário, devemos antes colocar o símbolo composto //. Tudo o que vier depois do // será totalmente ignorado pelo compilador; por-

tanto, podemos escrever qualquer coisa. Caso o comentário seja maior do que uma linha, como acontece nas linhas 9 e 19 do Código 2.4, cada linha deve conter o símbolo de comentário novamente.

O Java possui outra forma de colocar comentários que permite a inclusão de um texto de muitas linhas. Nesse outro modo delimitamos o texto com os símbolos /* e */. O símbolo /* inicia uma área de comentário que apenas irá terminar quando o Java encontrar o símbolo */ (Código 2.5).

Código 2.5 Exemplo de comentário

```
class jogo
{

/* ainda está muito cedo
para você querer entender como é um jogo...*/
}
```

O uso adequado de comentários é uma ciência muito valiosa. Para um humano ler o trecho de programa pode ser algo bastante doloroso e cansativo, especialmente se não for alguém muito experiente. Assim, é muito conveniente escrever comentários que permitam o leitor não precisar ler detalhadamente todo o programa para entender o que este está fazendo. Por outro lado, escrever comentários óbvios ou desnecessários também pode ser muito aborrecido. Como dissemos, o uso conveniente dos comentários é uma ciência valiosa e que irá amadurecer à medida que o programador for se tornando mais experiente.

2.4. Variáveis

As variáveis são um dos elementos fundamentais para poder se fazer um programa um pouco mais avançado do que o "Hello World". Aliás, sem o conceito de variáveis não poderemos fazer praticamente nenhum programa realmente útil.

As variáveis consistem em pedaços de memória ao qual damos um nome. Uma vez batizado, o usuário pode usar esse espaço de memória para armazenar os valores que quiser, resgatá-los a qualquer momento ou mudar o valor que lá foi escrito, apenas usando o nome indicado. Imagine um contador de vida: sempre que o jogador cometer algum erro o programa deverá ir a um espaço de memória, chamado `vidas`, ver quanto era o valor que estava escrito e mudá-lo por um novo número. O programador não precisará se preocupar

com o endereço físico de onde reside essa variável, apenas referenciar o nome escolhido.

Uma variável pode ser de vários tipos diferentes. A variável `vidas`, por exemplo, tem de ser um número inteiro, tal como 3. Já uma variável `saude` que guarda valores de energia de um jogador tem de ser do tipo que armazena números reais (com ponto decimal), tal como 0.32. Uma variável `nomeJogador` que armazena o nome do jogador deve ser do tipo "sequência de caracteres", tal como "Sheeva".

Para declarar uma variável (é assim que chamamos o processo de criar uma variável) devemos dizer qual é o tipo do dado que ela irá receber: números inteiros, números reais, palavras, imagens, sons etc. É necessário avisar isso ao compilador, para que ele saiba o tamanho adequado e a maneira correta de armazenar os dados. A declaração deve ter esta sintaxe:

```
<tipo_da_variável> <nome_da_variavel>;
```

Assim, para declarar uma variável do tipo inteiro devemos escrever:

```
int vidas;
```

O Java possui oito tipos primitivos, conforme a Tabela 2.1.

Tabela 2.1 Tipos primitivos definidos no Java

Nome	Descrição	Exemplos
byte	Um byte é uma composição de 8 bits, o que corresponde a 2^8 possibilidades de dados, sendo, portanto, possível representar valores numéricos inteiros que variam de 0 a 255, ou valores de –128 a 127.	3, -10, 128, 0
short	Corresponde a 2 bytes (16 bits), podendo representar valores numéricos inteiros de -32768 a 32767	-10000, 0, 10, 128, 20000
int	Corresponde a 4 bytes (32 bits), podendo representar valores numéricos inteiros de -2147483648 a 2147483647	-2000000000, 0, 11, 1913331112
long	Corresponde a 8 bytes (64 bits), podendo representar valores numéricos inteiros de –9223372036854775808 a 9223372036854775807	-2000000000000000000, 0, 1, 12345678
float	Tal como o tipo int, este utiliza 4 bytes da memória, porém os dados são armazenados de maneira que podemos representar números reais, portanto, com casas decimais. Um float permite números reais com até 7 dígitos e é capaz de representar valores com expoentes de até 10^{-45} ou 10^{38}.	-10.0, 0.1, 0.22, 3.1415926, 1.23456 e -12

Nome	Descrição	Exemplos
double	Da mesma forma que o float, neste tipo os bytes são formatados de maneira a suportarem números reais, porém aqui utilizam-se 8 bytes, podendo-se representar números com até 15 dígitos e é capaz de representar valores com expoentes de até 10^{-324} ou 10^{308}.	-1.0, 0.0, 3.1415926535897, 1.57 e -102

char	Este tipo, formado por 2 bytes, armazena um caractere simples, tal como A, a, t, *. Também pode conter um caractere especial, tal como uma quebra de linha ou um indicador de final de texto. Ao associar um caractere a esta variável, devemos usar o símbolo ´ para identificar que o caractere não é uma variável ou um operador:
	`char letra;`
	`letra = ´A´`
	Para poder escrever caracteres especias, devemos usar o simbolo \ seguido de algum identificador. Alguns exemplos:
	\n: quebra de linha
	\t: TAB
	\´: ´
		A, b, $, 4 (caractere 4, não o número 4).

boolean	Este é um tipo lógico, que pode apenas ser true ou false. Será usado para expressões lógicas e de relacionamentos	true, false

Há muitos outros tipos, mas de alguma forma são composições ou arranjos desses tipos primitivos. Vamos apresentar apenas mais um tipo: o **String**, que consiste numa sequência de caracteres (Tabela 2.2). Neste livro vamos manter o termo em inglês *string* em vez de sua tradução "cadeia de caracteres", não só por causa do seu largo uso entre programadores, mas também porque ajuda na familiarização com os ambientes Java.

Tabela 2.2 Tipo String do Java

Nome	Descrição	Exemplos

String	Consiste numa sequência de caracteres. Para que o programa não confunda essa sequência de caracteres com nomes de variáveis, colocamos o seu conteúdo entre aspas:
	`String mensagem;`
	`mensagem = "Apertem os cintos";`
	Caso não houvesse aspas, o compilador pensaria que as palavras `Apertem`, `os` e `cintos` seriam nomes de variáveis e procuraria o conteúdo das mesmas. Nesse caso, como as variáveis não foram declaradas, ocorreria um erro do tipo "variável Apertem não declarada".
	Tudo o que estiver dentro das aspas é tratado como um texto e podemos ficar tranquilos, pois o compilador não irá executar nada; apenas copiará esses caracteres para uma área de memória ou para a interface de saída.
		"Hello Word", "para o além e infinito", "dois", "1, 2, 3 e já!", "2"

O programador pode dar o nome que quiser para as variáveis, desde que obedeça a algumas pequenas restrições listadas a seguir:

- Os nomes não podem ser palavras reservadas do sistema. Exemplo: `int main`. Nesse caso, a palavra `main` já tem um significado predefinido pela linguagem, e ocorreria um conflito quando o Java encontrasse essa palavra no código.
- Os nomes devem começar com letras, mesmo que possa haver números no meio da palavra. `NPC1`, `fase2` ou `campo3d` são nomes possíveis, mas `2vidas` ou `123` não são nomes permitidos.
- Não pode haver espaço numa variável. Assim, ao declarar `int modo facil`, haverá um conflito, pois o Java entenderá que `modo` é uma variável inteira, mas `facil` estará solto e não fará parte do nome composto. Para colocar nomes compostos, há duas estratégias: separar as palavras por letras maiúsculas ou separar pelo símbolo ´_´ (chamado de *underscore*, em inglês). Dessa maneira, `modoFacil` e `modo_facil` são nomes válidos para variáveis.
- As letras maiúsculas e minúsculas são consideradas caracteres diferentes pelo Java. Dizemos que o Java é **sensível a caixa** (***case sensitive***). Assim, as variáveis `Velocidade` e `velocidade` são totalmente diferentes.

Além das regras formais para dar nomes a variáveis, há algumas pequenas regras de "boas maneiras" que ajudam para uma programação "saudável":

- Usar nomes de variáveis sugestivos. Um programa poderá conter centenas ou até milhares de variáveis; portanto, se os nomes não ajudarem a lembrar o que significam, o programador poderá perder muito tempo tentando recordar qual era o sentido inicial. `X2, i3` e `zzz` não são nomes muito sugestivos, mas `tempo_total, contadorDeTempo` e `Estado_do_personagem` são nomes mais interessantes.
- Como é muito comum o programador usar nomes de variáveis compostos, é interessante que ele use um padrão constante a respeito de como separar as palavras, por exemplo: usar sempre o símbolo "_" ou usar a separação com letras maiúsculas, começando a primeira em minúscula (`tempoTotal`, `contadorDeVidas` etc.). Esse modo é o mais usual e é chamado de "**Camel Case**", devido às diversas corcundas formadas pelas maiúsculas no meio do corpo.
- Excetuando o símbolo "_", não é conveniente usar nenhum outro símbolo. Nunca use acentos.

- Apesar do Java não impor restrições sobre o tamanho máximo do nome de uma variável, é bom que o programador seja econômico, uma vez que a chance de errar na tipografia de palavras grandes é maior do que em palavras menores.

Há outras regras que podem ser adotadas, porém estão além do escopo deste livro e devem ser objeto de consideração apenas quando você se tornar um programador experiente. Uma delas, chamada de **Notação Húngara**, é tão interessante quanto polêmica. Nela usamos determinadas letras ou conjunto de letras minúsculas como prefixos para indicar os tipos das variáveis. Dessa maneira, quando estamos usando uma variável numa posição do código bem longe do local onde foi declarada, sabemos qual é o seu tipo. Por exemplo, o prefixo `i` denota uma variável inteira, `f` denota uma variável float e `str` indica um String: `iContador`, `fSaude` e `strCancelar`. A notação húngara também nos ajuda a identificar quando estamos fazendo atribuições entre tipos diferentes, por exemplo: `iContador = iNivel` e `iContador = fNivel`. Entretanto, por outro lado, a notação Húngara sobrecarrega a legibilidade do nome e é tediosa.

Depois que uma variável é declarada, você pode atribuir um valor a ela usando o operador de atribuição (*assignment operator*) "`=`". A sintaxe para o comando de atribuição é a seguinte:

```
variável = expressão;
```

onde `expressão` pode ser um número, uma expressão matemática ou até mesmo outra variável.

Ao fazer uma atribuição de valor desse tipo, estamos na verdade escrevendo o valor desejado na posição da memória referenciada pela variável. Caso houvesse alguma coisa já escrita anteriormente nesse espaço de memória, o valor anterior seria perdido para sempre. Podemos declarar uma variável e atribuir um valor a ela em um mesmo comando, por exemplo:

```
int vidas = 3;
```

Em Java, tal como em outras linguagens, uma variável nasce de um tipo e deve permanecer desse tipo até a sua morte (isto é, até o fim do programa). Devemos ter cuidado com os tipos na hora das atribuições de valores. Por exemplo, não podemos atribuir um *string* a uma variável numérica do tipo `float`. Entretanto, é possível atribuir um valor a uma variável numérica cujo tipo suporta uma faixa maior de valores. Por exemplo, podemos atribuir um valor `int` para um valor `float`, isto é:

```
int k = 4;
float f = k;
```

A faixa de tipos numéricos cresce na seguinte ordem:

byte – short – int – long – float – double

Por outro lado, não podemos fazer o inverso, isto é: não podemos encaixar um tipo com faixa maior de valor em um de faixa menor. Esse último caso só é possível através de uma **operação de conversão de tipo** (***type casting***, em inglês) que consiste em anteceder o valor ou a variável com o tipo desejado entre parênteses. Por exemplo:

```
double d = 5.3;
int k;
k = (int)d; // k recebe o valor 5
```

Nesse exemplo, é importante notar que a variável não muda com o *casting* (`d` continua do tipo `double` e com o valor 5.3).

As linguagens de programação também têm a seguinte convenção comum para operações com números inteiros do tipo x/y, em que um ou os dois operandos são do tipo `int` e `/` é o operador de divisão:

- se os dois operandos são do tipo `int`, a divisão é feita com a precisão inteira. Por exemplo: 7/3 resulta em 2
- se apenas um dos operandos é do tipo `int`, o tipo `int` é automaticamente convertido no outro tipo. Por exemplo: 7.0/3 resulta na dízima 2.333...

Essa convenção faz parte do que chamamos de trabalhar com **precisão inteira**. Nesse tipo de trabalho também podemos incluir o **operador resto** (***remainder***), por exemplo: 7 % 3 retorna o resto 1.

Outros exemplos de atribuição de variáveis podem ser encontrados no Código 2.6. Observando a linha 2 desse código, podemos notar que duas variáveis estão sendo declaradas na mesma linha. Isso é possível desde que as duas sejam do mesmo tipo. Entretanto, não se deve abusar desse atalho, já que o código pode começar a ficar difícil de ser lido e comentado.

Código 2.6 Exemplos de associação de variável

```
1   int minuto;
2   int hora, dia;
3
4   int totalSegundos;
5
6   minuto = 60;
7   hora = minuto * 60;
8   dia = hora * 24;
```

Código 2.6 Exemplos de associação de variável(cont.)

```
totalSegundos = dia;   // Neste caso, resultado está recebendo o
                       // conteúdo da variável dia
System.out.print ("Um dia tem: ");
System.out.print (totalSegundos);
System.out.print (" segundos");
```

2.5. Entrada de dados

A tarefa de ler dados do mundo externo pode parecer trivial num primeiro momento. De fato, ela é simples, pois o Java encapsula (esconde) uma série de tarefas complexas, tais como mapear o teclado, converter o código da tecla lida para um valor correspondente, converter esse valor lido para o tipo desejado, reconhecer que a tecla ENTER foi pressionada e que, portanto, acabou o processo de leitura.

Para efetuar a leitura, temos de criar um objeto que possui as habilidades necessárias para essa comunicação. Existem diversas classes que trabalham com entrada de dados, sendo uma das mais comuns a classe *Scanner*. Como já foi mencionado, quando vamos usar um objeto num programa, não precisamos nos preocupar em ter de saber como ele funciona por dentro, a não ser que queiramos melhorá-lo ou modificá-lo. Assim, um objeto do tipo *Scanner* encapsula para nós todas essas tarefas complexas que envolvem uma leitura. Para podermos criar um objeto desse tipo, é preciso dizer ao compilador Java para que traga uma biblioteca ou trecho de código que diga como funciona essa classe. Fazemos isso através do comando `import`, tal como pode ser visto na linha 1 do Código 2.7:

```
import java.util.Scanner;
```

Uma vez que uma classe é importada, podemos fazer tudo como se o código da classe em questão estivesse presente no programa que estamos fazendo.

Ao realizar uma leitura de dados teremos de armazenar o valor lido em algum local, para que depois possamos usar esse valor como bem entendermos. Na maioria das vezes atribuiremos o resultado da leitura a uma variável. Outro detalhe importante ao fazer uma leitura consiste em dizer de que tipo é o valor que estamos lendo: pode ser um caractere, um número inteiro, número real ou até um texto, do tipo `String`. Dessa forma, antes de efetuar a leitura teremos de criar uma variável que seja do mesmo tipo da leitura. No Código 2.7, estamos fazendo a leitura de um nome. Portanto, `nome` deve ser do tipo `String`.

Código 2.7 Programa Ler para leitura de dados

```
1   import java.util.Scanner;
2   public class Ler
3   {
4      public static void main(String[] args)
5      {
6         String nome;
7         Scanner in;
8         in= new Scanner(System.in);
9         System.out.println(”Entre com o seu nome”);
10        nome = in.next();
11        System.out.println(”Bem vindo, " + nome);
12     }
13  }
```

Para criar um objeto que seja do tipo da classe desejada, usamos o construtor da classe. Um construtor é um método que cria objetos da sua classe. No Código 2.7, o construtor `Scanner()` cria um objeto da classe `Scanner`. Construtores sempre têm o mesmo nome da classe. Os construtores serão vistos com mais detalhes em seções posteriores deste livro, mas, por agora, basta que você consiga interpretar as linhas 7 e 8: primeiro declaramos o objeto `in` como sendo do tipo `Scanner` (linha 7) e a seguir criamos esse objeto usando o operador `new` (linha 8). Essas duas linhas poderiam ter sido reunidas em um único comando:

```
Scanner in = new Scanner(System.in);
```

Antes de efetuar a leitura, avisamos ao usuário que ele deverá entrar com um determinado valor. Essa tarefa é importante, caso contrário, o usuário verá uma tela em branco e não saberá o que deve fazer, pois o programa fica parado até que o usuário digite algo no teclado e aperte a tecla ENTER. A linha 10 realiza a leitura propriamente dita, através do método `next()` da classe `Scanner`. O método `next()` lê um *string* (que não pode ter espaços em branco). O resultado do valor digitado é armazenado na variável `nome`, que na linha 11 é utilizada para uma impressão. O objeto `in`, que é um objeto do tipo `Scanner`, tem diversos métodos associados a si. Nesse caso, estamos usando o método `next()` que realiza a leitura de um texto, assumindo que o resultado estará sendo atribuído a uma variável de tipo `String`. Se `nome` fosse do tipo `int`, ocorreria uma incompatibilidade de tipos, e o programa não seria capaz de ser compilado. Para tanto, a classe `Scanner` possui uma função para cada tipo primitivo existente no Java: `nextBoolean()`, `nextByte()`, `nextInt()`, `nextFloat()`, `nextLong()` e `nextDouble()`.

A linha 11 do Código 2.7 imprime o resultado da leitura. Perceba que há uma concatenação de dois *strings*, através do operador `´+`` (que tanto serve para somar dois números como para concatenar dois strings): `"Bem vindo, " + "Buzz"` resulta em `"Bem, vindo, Buzz"`. Como o método `System.out.println` tem como argumento apenas um *string*, em vez de chamarmos o método várias vezes, realizamos uma concatenação de dois *strings* e pedimos para imprimir esse resultado. Se um dos operandos é numérico, há uma conversão automática de número para o tipo `String` antes da concatenação. Por exemplo, se

```
String a = "Dia ";
int k = 26;
```

então `a + k` resulta em `"Dia 26"`.

2.6. Expressões

Uma expressão matemática é um trecho de programa que computa o resultado de uma série de operações concatenadas. Se o resultado de uma expressão é numérico, chamamos essa expressão de **expressão aritmética**. Se o resultado for um valor lógico, temos uma **expressão lógica**. Os elementos básicos de uma expressão são os operandos e os operadores.

Alguns exemplos de operandos: 5, 3.1416, 0

Exemplos de operadores: +, –, *

Exemplo de uma expressão aritmética: (3+7)*(2+5)

Quando um programa encontra uma expressão, tenta resolvê-la substituindo imediatamente o espaço que era ocupado pela expressão com o seu resultado correspondente. Caso não consiga resolver, o programa inteiro irá parar, já que o computador não tolera que uma expressão fique sem ser resolvida.

Uma expressão pode ser composta de outras expressões, que, por sua vez, podem possuir outras expressões. Chamamos estas de **Expressões Compostas**.

Tal como nós, seres humanos, ao receber uma expressão para ser calculada, o computador terá de fazer o cálculo por partes. No exemplo (3+7)*(2+5), o computador fará primeiro as expressões de soma que estão entre parêntesis e posteriormente resolverá a multiplicação.

Uma expressão pode ter variáveis como operandos numéricos, desde que as variáveis sejam de tipos numéricos, tal como pode ser visto no Código 2.8.

Código 2.8 Exemplos de expressões

```
1   double raiz1, raiz2;
2   double delta;
3   double a, b, c;
4   a = 3;
5   b = 4;
6   c = 1;
7
8   delta = b*b – 4*a*c;     // delta recebe o número 2
9   raiz1 = (-1*b + Math.sqrt(delta)) / (2*a);
10  raiz2 = (-1*b – Math.sqrt(delta)) / (2*a);
```

Na linha 8 do Código 2.8, o compilador irá primeiro resolver a expressão que está do lado direito da variável `delta`, para depois realizar a atribuição do resultado final da expressão à variável `delta`. Dentro de uma expressão, caso um método retorne um valor numérico, esse método também pode ser considerado um operando. Isso ocorre na linha 9 do Código 2.8, em que o método `sqrt()` da classe `Math` retorna a raiz quadrada de `delta`.

Costumamos apresentar os métodos com os tipos de seus argumentos (e quando possível com nomes sugestivos, tais como `radianos`) e com o tipo de valor que retornam. Dessa maneira, o método que extrai a raiz quadrada é apresentado da seguinte forma:

```
double sqrt(double x)
```

Cabe ao usuário lembrar que a chamada desses métodos exige a indicação da classe, por exemplo:

```
Math.sqrt()
```

Alguns dos métodos mais usados da classe `Math` são os seguintes:

`double sin(double radianos)`	*seno(radianos)*
`double cos(double radianos)`	*coseno(radianos)*
`double tan(double radianos)`	*tangente(radianos)*
`double toRadians(double graus)`	*converte graus para radianos*
`double toDegrees(double radianos)`	*converte radianos para graus*
`double pow(double x, double a)`	*potência* x^a
`double log(double x)`	*logaritmo natural ln(x)*
`double log10(double x)`	*logaritmo base 10 log(x)*
`long round(double x)`	*arredonda número real para o inteiro mais próximo*

Um método da classe `Math` muito poderoso e útil para jogos é o que gera um valor aleatório x no intervalo [0.0, 1.0), isto é, $0 \leq x < 1.0$:

```
double random()
```

Você pode usar esse método para gerar números aleatórios em qualquer intervalo. Para tanto use a seguinte expressão:

```
a + Math.random() * b
```

que retorna um número aleatório $x \in [a, a + b)$, isto é: $a \leq x < a + b$.

Você não deve confundir variáveis com constantes. Muitas vezes queremos usar uma variável que representa uma grandeza imutável (por exemplo, a aceleração da gravidade 9.807 m/s^2). Como podemos garantir que uma variável que a representa jamais terá o seu valor alterado? Para isso definimos uma **constante** com a seguinte sintaxe:

```
final tipo NOME = valor;
```

onde o nome da constante é, por convenção (mas não obrigatório), escrito com letras maiúsculas. A constante deve ser declarada e inicializada no mesmo comando. A palavra-chave **final** significa que a constante não pode ser alterada. O código a seguir ilustra o uso de constantes:

```
public class Main
{
   public static void main(String[] args)
   {
      final double G = 9.807;  // constante
      double deslocamento, t = 1;
      deslocamento = (G * t * t)/2;
      System.out.println(deslocamento);  // exibe: 4.9035
   }
}
```

Tabela 2.3 Precedência de operadores do Java

1) Parêntesis:	()
2) Operadores Unários:	++, --,!, unário – e +, type-cast
3) Multiplicação e Divisão:	*, /, %
4) Adição e Subtração:	+, -
5) Operadores Relacionais:	<, >, <=, >=
6) Igualdade e Desigualdade:	==,!=
7) E Booleano:	&&
8) OU Booleano:	||
9) Operador Condicional:	?:
10) Operadores de Atribuição:	=, +=, -=, *=, /=, %=

Dentro de uma expressão aritmética há uma certa prioridade entre os operadores. Na linha 8 do Código 2.8, há vários arranjos que poderiam ser imaginados para calcular o valor para `delta`, tais como

```
b*(b-4)*(a*c)
((b*b)-(4*a))*c)
```

Cada uma dessas expressões produz um resultado diferente. Entretanto, a expressão do Código 2.8 não gera ambiguidade, pois sabemos que a multiplicação tem prioridade maior que a subtração, portanto, primeiro são calculados os termos de multiplicação `b*b` e `4*a*c` para posteriormente realizar a subtração entre o resultado das duas subexpressões. Assim sendo, as linguagens de programação procuram seguir uma lista de prioridades entre os operadores, tal como ocorre na matemática. Chamaremos essa lista de "tabela de precedência". A Tabela 2.3 mostra a lista de prioridades do Java. Você não deve se preocupar com os operadores que não conhece, pois serão apresentados mais adiante.

Se um dia o seu chefe lhe der a boa notícia: "vou incrementar o seu salário", acredito que você irá entender que ele verá qual é o valor atual do seu salário e somará um valor extra, para logo em seguida o salário ter um novo valor. Se `salario` for uma variável, podemos escrever esse processo da seguinte maneira:

```
salario = salario + 1000;  // Supondo que o incremento do salário seja de R$ 1000...
```

Esse tipo de operação é muito comum em computação (...não estamos nos referindo ao aumento de salário) e recebe o sugestivo nome de ***incremento* de variáveis**. Essa operação não se usa na matemática e você, iniciante em programação, deve certamente tê-la estranhado um pouco. É importante notar que a expressão do lado direito tem prioridade maior que a atribuição (basta consultar a Tabela 2.3). Assim sendo, o Java primeiro irá efetuar o cálculo de `salario + 1000`, para depois realizar a atribuição. Estamos colocando o resultado na própria variável `salario`, que passará a conter um novo valor, perdendo o valor original. Não é necessário nos estendermos em demasia para dizer que decremento de variáveis é algo semelhante: a variável será subtraída de um valor e esse resultado será armazenado novamente na mesma variável, perdendo-se o valor que havia originalmente nela. Por exemplo:

```
vidas= vidas-1; // O jogador cometeu algum erro e estamos tirando uma de suas vidas...
```

O mesmo vale para os outros operadores aritméticos:

```
x = x * 5;
x = x / 5;
x = x % 5;
```

Código 2.9 Incremento e decremento de variáveis

```
1   double tempo, velocidade;
2
3   velocidade = 30;
4   tempo = 100;
5
6   velocidade = velocidade + 10; // A velocidade vale 40 neste ponto
7   velocidade += 15;             // A velocidade agora vale 55
8   velocidade++;                 // Agora a velocidade vale 56
9
10  tempo = tempo - 5;            // tempo passa a valer 95
11  tempo -= 10;                  // agora o tempo vale 85
12  tempo--;                      // tempo vale 84
```

O processo de incremento e decremento de variáveis é tão comum em programação que quase todas as linguagens nos oferecem alguns operadores especiais que funcionam como atalhos simplificados (também conhecidos pelo termo ***shorthand operators***). As linhas 7 e 11 do Código 2.9 apresentam o primeiro tipo de atalho para escrever incremento e decremento. Tratam-se dos **operadores de atribuição compostos**, cuja sintaxe é a seguinte:

```
variável <operador aritmético>= <valor do incremento>
```

Por exemplo:

```
x += 5;
x -= 5;
x *= 5;
x /= 5;
```

Esses operadores não possuem nenhuma diferença em relação ao procedimento original de incremento/decremento de variáveis. No entanto, eles podem ser considerados atalhos porque permitem que coloquemos do lado direito apenas o valor do incremento ou decremento a ser efetuado. Outros exemplos de operadores de atribuição compostos podem ser encontrados no Código 2.10.

Código 2.10 Operadores *= e /=

```
municao = 3;
municao *= 10;  // equivale a municao = municao * 10
                // Neste caso municao passa a valer 30

saude = 10;
saude /= 2;     // equivale a saude =  saude / 2
                // Neste caso a saude foi diminuida pela metade...
```

As linhas 8 e 12 do Código 2.9 apresentam atalhos mais resumidos ainda. Tratam-se dos **operadores de incremento e decremento ++ e --**, cuja sintaxe é a seguinte:

forma prefixada:

```
++<variável>;
--<variável>;
```

forma pós-fixada:

```
<variável>++;
<variável>--;
```

Se o operador está *antes* da variável (forma prefixada), a variável é incrementada ou decrementada de 1 e o novo valor da variável é retornado. Se o operador está *depois* da variável (forma pós-fixada), o valor antigo e original da variável é retornado e depois a variável é incrementada ou decrementada de 1. No caso de não haver atribuição (como nas linhas 8 e 12 do Código 2.9), não há diferença entre as formas prefixada e pós-fixada. Entretanto, o comportamento é completamente diferente quando temos atribuição. Por exemplo, se temos:

```
int n = 3;
int vidas = 0;
```

o seguinte comando atribui o valor 3 a `vidas` e depois incrementa o valor de `n` (que passa a valer 4):

```
vidas = n++;
```

porém o seguinte comando incrementa `n` de 1 (que passa de 3 para 4) e depois atribui o novo valor à variável `vidas`:

```
vidas = ++n;
```

Note que, nesses dois exemplos, a variável `n` é incrementada de 1 (isto é, `n` passa a valer 4).

Uma expressão é dita algébrica ou numérica quando o resultado que se espera dela é um valor algébrico ou numérico. Também podemos ter expressões relacionais e expressões lógicas.

Uma **expressão relacional** é aquela que tem operadores relacionais (também chamados de operadores de comparação). Operadores relacionais comparam dois valores e retornam um valor Booleano (`boolean`) que pode ser `true` (verdadeiro) ou `false` (falso). Os operadores relacionais são os seguintes:

menor	`<`
maior	`>`
igual	`==`
diferente	`!=`
menor ou igual	`<=`
maior ou igual	`>=`

Código 2.11 Exemplos de expressões relacionais

```
public class Main
{
   public static void main(String[] args)
   {
      int vidas = 3;
      int n = 1;
      int k = 1;
      double lado = 10;
      double posicaoX = 5;
      double posicaoY = Math.sqrt(lado);

      System.out.println(vidas <= 0);
      System.out.println(n == k);
      System.out.println(Math.sqrt(lado) < posicaoX);
      System.out.println(posicaoY < 2.0);
   }
}
```

O Código 2.11 contém alguns exemplos de expressões relacionais. Analise-o cuidadosamente, observando que podemos comparar variáveis e expressões. Se você rodar este programa, obterá o seguinte resultado:

```
false
true
true
false
```

Cuidado para não confundir o operador de atribuição `=` com o operador relacional `==`.

Os operadores relacionais também podem comparar caracteres, porque os caracteres são representados por um código padrão que permite comparações (são equivalentes a um número inteiro). Dessa maneira, a letra `a` é menor do que `b`, mas é maior do que a sua versão maiúscula `A` (as letras maiúsculas vêm primeiro no código padrão). Por exemplo:

```
char letra1 = ’a’;
char letra2 = ’A’
System.out.println(letra1 > letra2); // exibe o resultado true
```

Aliás, os operadores de incremento e decremento podem ser usados em variáveis `char`, por exemplo:

```
char letra = ’a’;
System.out.println(++letra); // exibe o próximo caracter, que é o b
```

Você pode fazer um *casting* para saber o número correspondente a um determinado caractere (procure conhecer as tabelas ASCII e Unicode na Internet, para entender melhor o que isso significa). Os valores numéricos para `‘a’`, `‘b’` e `‘A’` são 97, 98 e 65. Tente, por exemplo, o seguinte:

```
int iLetra = (int)'b';
System.out.println(iLetra); // exibe o resultado 98
```

Uma observação importante: operadores relacionais não comparam *strings*. Para isso você precisa usar um método especial, o que será visto no Capítulo 6.

Devemos observar, na tabela de precedência de operadores (Tabela 2.3), que as operações relacionais possuem menor prioridade do que as operações algébricas. Isso significa que primeiro são resolvidas as expressões aritméticas para posteriormente serem avaliadas as relações entre os valores.

Os **operadores lógicos**, também chamados de **operadores Booleanos**, tais como `&&` para **E** e `||` para OU, operam sobre valores Booleanos para criar um novo valor Booleano. Operadores relacionais e lógicos podem ser combinados, por exemplo:

<u>se</u> `(vidas <= 0) || (saude == 0)` <u>então</u> avise que o jogador morreu

A expressão `(vidas <= 0) || (saude == 0)` significa "`vidas` menor ou igual a zero OU `saude` igual a zero". Infelizmente, no presente momento, é provável que você não saiba como implementar a construção "<u>se</u> ... <u>então</u> ...". Isso será visto no próximo capítulo. No entanto, você pode testar o seguinte código:

```
int vidas = 1;
int saude = 0;
System.out.println((vidas <= 0) || (saude == 0)); // exibe o resultado true
```

Os operadores lógicos são os seguintes:

operadores binários

`&&`	significa **E**	exemplo: `a && b`
`||`	signifca **OU**	exemplo: `a || b`
`^`	significa **OU EXCLUSIVO**	exemplo: `a ^ b`

operador de um lugar

`!`	significa **NÃO**	exemplo: **!**`a`

e as expressões lógicas têm a seguinte sintaxe:

```
<expressão_Booleana> <operador_lógico_binário> <expressão_Booleana>
<operador_lógico_de_um lugar> <expressão_Booleana>
```

O exato funcionamento dos operadores lógicos pode ser resumido na chamada **Tabela Verdade** (Tabela 2.4), que define os possíveis resultados Booleanos a partir das expressões Booleanas `a` e `b`.

Tabela 2.4 Tabela Verdade para os operadores lógicos

a	b	!b	a && b	a || b	a ^ b
false	false	true	false	false	false
false	true	false	false	true	true
true	false	true	false	true	true
true	true	false	true	true	false

Na Tabela 2.4 fica claro que a expressão `a && b` só é verdadeira (`true`) quando as duas expressões são verdadeiras ao mesmo tempo. Nessa tabela, também fica claro que `a || b` é verdadeira quando uma das expressões é verdadeira ou quando as duas o são (só é falsa quando as duas são falsas ao mesmo tempo). Já no **OU EXCLUSIVO**, temos que `a ^ b` só é verdadeira se uma das expressões for verdade e a outra não (isto é, as duas não podem ser verdadeiras ao mesmo tempo). Quando você diz para alguém "você joga comigo ou é meu inimigo", o que quer dizer? Você está usando que tipo de OU?

O Java, por questões de eficiência, tem um esquema especial para avaliar as expressões Booleanas. Por exemplo, na expressão `a && b`, o Java primeiro avalia `a` e se acontecer de `a` ser falsa, o Java abandona a avaliação de `b` e já retorna o resultado `false`. No caso de `a || b`, o Java primeiro avalia `a` e se acontecer de `a` ser verdadeira, o Java abandona a avaliação de `b` e já retorna o resultado `true`. Se o programador quiser forçar a avaliação da expressão que o Java eventualmente pode abandonar, deve usar os **operadores lógicos incondicionais** `&` e `|` que obrigam a avaliação das duas expressões, em quaisquer condições. Por exemplo:

```
(vidas > 0) & (saude-- < 3)
```

significa que `saude` sempre será decrementada independente de `vidas` ser maior do que 0 ou não, pois a segunda expressão sempre será executada.

Algumas vezes vamos querer armazenar o resultado de uma expressão lógica numa variável. Nesse caso, a variável não pode ser inteira nem real, porque o conteúdo que se deseja armazenar não é um número, mas sim um resultado lógico (falso ou verdadeiro). Para tanto, existe um tipo `boolean`, a cujas variáveis só podemos atribuir `true` ou `false`. Observe o Código 2.12 para alguns exemplos.

Código 2.12 Exemplos de expressões lógicas compostas

```
public class Main
{
   public static void main(String[] args)
   {
      boolean fim;
      boolean parado;
```

Código 2.12 Exemplos de expressões lógicas compostas (cont.)

```
        boolean ataque;
        int x = 5;   // coordenada X em pixels
        int y = 25;  // coordenada Y em pixels
        int vidas = 3;
        int saude = 50;
        int velocidade = 0;
        fim = (vidas == 0) || (saude <= 0);
        parado = (velocidade == 0);
        ataque = (X > 0) && (X < 10) && (Y > 20) && (Y < 30);
        System.out.println("fim "+fim+"\n"+"parado "+parado+"\
        \n"+"ataque "+ataque);
    }
}
```

EXERCÍCIOS

E2.1. Liste todas as variáveis com seus respectivos tipos que você imagina serem necessários para implementar o jogo Pong.

E2.2. Observe a seguinte regra da aeronáutica (é uma regra fictícia, provavelmente não seja tão simples assim...) “para que um avião possa decolar de um aeroporto muitas condições devem ser satisfeitas: caso a pista seja menor que 1,5 km o avião deve pesar menos que 40 toneladas, caso a pista possua uma medida entre 1,5 km e 2 km, o avião não pode ultrapassar das 60 toneladas. Caso a pista seja maior que 2 km, qualquer avião pode decolar. Caso a visibilidade seja menor do que 20 metros, apenas aviões com sistema de decolagem computadorizada podem decolar. Se estiver chovendo acima de 5mm, nenhum avião pode decolar. Se o avião possuir mais de 100 passageiros e estiver chovendo, independentemente da intensidade, o avião não pode decolar”. Escreva uma expressão lógica para retratar essa situação, armazenando o resultado na variável booleana “decola”. As variáveis já existentes são: tamanhoDaPista (float), pesoDoAviao (float), visibilidade (inteiro), intensidadeDaChuva (inteiro), numeroDePassageiros (inteiro), possuiSistemaDeDecolagem (booleano).

E2.3. Faça um programa que solicita o raio de uma circunferência e imprime sua área e seu perímetro.

E2.4. Faça um programa que leia uma temperatura em Celsius e a converta para Fahrenheit.

E2.5. Faça um programa que imprima um menu semelhante a este:

1 – Iniciar Jogo

2 – Restaurar último Jogo

3 – Configurar Jogo

4 – Sair

Em seguida, o programa deverá ficar esperando pela entrada de uma opção e imprimir na tela qual foi a opção digitada.

E2.6. Escreva um programa que receba a coordenada superior esquerda de um retângulo, bem como a largura e altura do mesmo e em seguida imprima as coordenadas dos quatro vértices que o compõem. Assuma que os lados do retângulo estão alinhados com os eixos.

E2.7. Dizemos que um quadrado envolvente de um círculo é o menor quadrado possível que contenha um círculo em seu interior. Assumindo que esse quadrado está alinhado com os eixos, faça um programa que receba o canto superior esquerdo e o canto inferior direito do quadrado e imprima qual é a coordenada do centro do círculo, bem como o seu raio.

E2.8. Sabendo que a velocidade do som é de 340 m/s, escreva um programa para calcular a distância de um trovão até uma pessoa. Para tanto, a pessoa deve entrar no programa com o tempo que transcorreu desde o momento que viu o clarão do trovão até o momento em que ouviu o seu som. O tempo pode ser um valor real, uma vez que a pessoa pode usar um cronômetro de alta precisão.

E2.9. Escreva um programa que recebe os três vértices que compõem um triângulo e imprime o seu perímetro e a sua área.

E2.10. A cada frame de um jogo, dada uma intensidade de força para um corpo rígido capaz de se mover, bem como o tempo transcorrido do frame anterior para o corrente, deve-se calcular quanto o objeto se deslocou. Faça um programa que leia o instante inicial e final de um frame (em ponto flutuante), a velocidade inicial de um corpo que está em queda livre, sua altura em relação ao solo (e centímetros) e calcule qual deverá ser a nova altura do mesmo para o frame seguinte. Assuma a força da gravidade como sendo a constante 9,8.

E2.11. Um vetor 3D pode ser representado por três valores reais *x, y* e *z*. Faça um programa que leia esses valores e calcule o tamanho do vetor.

Capítulo

3

Controle de Fluxo por Comandos de Seleção

O controle de tráfego de carros consiste em colocar condições, de forma que dezenas ou centenas de carros possam transitar com certa harmonia, evitando confusão, ou pelo menos tentando evitar... Parte desse controle consiste basicamente em colocar condições para que os motoristas parem ou andem: se o sinal estiver vermelho, deve-se parar; se estiver verde, pode seguir em frente. Se um pedestre atravessar a rua, o motorista deve parar e dar-lhe prioridade. Um programa pode ser comparado a um tráfego, porém não de carros, mas sim de dados.

Assim, se quisermos controlar o movimento de um personagem através de teclas, teremos de fazer uma leitura dos dados de entrada e em seguida testarmos se a tecla '→' foi pressionada. Nesse caso teremos de mover o personagem para a direita, mas se a tecla foi a '←', então teremos de mover o personagem para a esquerda. Obviamente poderemos ter um controle de fluxo mais complexo e ter de decidir movimentos do personagem para muitas outras teclas.

Todas as linguagens de programação, por mais simples e primitivas que sejam, devem possuir comandos para controle de fluxo, caso contrário, não seria possível fazer muita coisa diferente do "Hello World" e talvez os computadores não estivessem tão presentes em nossas vidas.

Este capítulo trata de controle de fluxo por comandos de seleção. No próximo capítulo veremos uma outra forma ainda mais poderosa de controle.

3.1. O comando if-else

O comando mais importante para controle de fluxo por seleção é o if / else. Sua sintaxe é apresentada no Código 3.1. Tão comum é esse comando que provavelmente você não verá nenhum programa ou trecho de código a partir de agora sem utilizá-lo.

Código 3.1 Comando if

```
if (expressão lógica)
   Comando 1     // caso verdade
else
   Comando 2    // caso falso
```

Podemos interpretar esse comando da seguinte maneira: se a expressão lógica for verdadeira, então execute o comando 1, caso contrário execute o comando 2, que corresponde ao caso falso. Como é óbvio, nunca serão executados os comandos 1 e 2 ao mesmo tempo, pois nunca uma expressão pode ser verdadeira e falsa ao mesmo tempo. De alguma forma, podemos afirmar que estamos ensinando o computador a tomar uma decisão, para uma dada situação, descrita pela expressão lógica.

Como foi visto no Capítulo 2, uma expressão lógica deve retornar um valor de verdadeiro ou falso (true/false). Assim, a primeira coisa que o computador irá fazer é avaliar essa expressão lógica usada na sintaxe do if. Ela pode ser muito simples e pequena, mas também pode ser complexa, utilizando operadores lógicos, como vistos no capítulo anterior.

Nem sempre será necessário que haja comandos para o caso verdade e para o falso. Pode ocorrer de apenas querermos executar algo no caso de a expressão ser verdadeira, mas caso ela seja falsa simplesmente querermos continuar o fluxo do programa, como se não existisse o comando if. Veja o Código 3.2:

Código 3.2 Comando if sem o else

```
1    If (tempo < 10)
2       System.out.print(”Seu tempo está acabando...”);
3    tempo = tempo -1;
```

O Código 3.2 implementa um controle de fluxo que, apenas no caso de o tempo ser menor do que 10 segundos, será impressa a mensagem “Seu tempo está acabando”. Indepentendentemente de o tempo ser maior ou menor do que 10, ele será decrementando, conforme a linha 3.

Os comandos 1 e 2 do Código 3.1 podem corresponder na verdade a vários comandos, não sendo restritos a apenas uma instrução. Para se colocar uma sequência de dois ou mais comandos, deve-se criar um bloco, usando os símbolos { e }, conforme pode ser visto no Código 3.3:

Código 3.3 Comandos compostos dentro do if

```
if (numero < 0)
{
   System.out.print(”o numero é negativo...”);
   numero = numero - 1;
}
else
{
   System.out.print(”o numero é zero ou positivo...”);
   numero = numero + 1;
}
```

Esse trecho de programa verifica se um número é positivo ou negativo. Caso seja negativo, o programa irá avisar o fato através de uma mensagem, e posteriormente irá decrementar o número. Caso seja positivo, também irá avisar, porém dessa vez irá incrementar o valor. Obviamente não há limites quanto ao número de comandos que podem ser colocados dentro de um bloco, sendo possível inserir até outros comandos de controle de fluxo.

Existem determinadas situações em que será necessário verificar uma condição que pode ter mais do que duas respostas, e, nesse caso, um simples if-else não será capaz de resolver o problema. Para tanto, será necessário utilizar uma sequência de controles de fluxo. O Código 3.4 exemplifica esta situação: queremos testar se um número é positivo, negativo ou zero. Para tanto, testaremos em primeiro lugar se o número é zero. Se for, avisamos o fato, e o problema está acabado. Se não for, ainda não somos capazes de dizer se o número é positivo ou negativo, o que nos levará a ter de fazer um novo teste.

Código 3.4 Sequências de if-else-if

```
if (numero == 0)
   System.out.print(”o numero é zero...”);
else
{
   if (numero > 0)
      System.out.print(”o numero é positivo...”);
   else
      System.out.print(”o numero é negativo...”);
}
```

Neste exemplo, apenas testaremos se o número é positivo ou negativo se ele for diferente de zero, uma vez que esse segundo teste reside dentro do escopo do else. Há uma outra forma de se construir controles de fluxo compostos, conforme exemplificado pelo Código 3.5:

Código 3.5 Sequências de if

```
if (numero == 0)
   System.out.print(“o numero é zero...”);
if (numero > 0)
   System.out.print(“o numero é positivo...”);
if (numero < 0)
   System.out.print(“o numero é negativo...”);
```

Nesse caso, mesmo o número sendo igual a zero, o programa irá testar se ele é positivo e logo depois se ele é negativo, já que todos os testes residem fora do corpo do if e se comportam como comandos independentes um em relação ao outro.

Em algumas situações, a sequência de if-else-if será mais conveniente e em outras a sequência de if será melhor. Nos casos de os testes serem exclusivos, ou seja, nunca um teste pode ser verdade em duas situações diferentes, o primeiro caso é melhor, já que estaremos economizando tempo, não tendo de continuar a efetuar os testes quando encontrarmos uma resposta verdadeira. Isso é o que ocorre no exemplo anterior, em que nunca um número poderá ser maior que zero, zero ou menor que zero ao mesmo tempo. Assim, quando o número for zero, o comando correspondente é executado e o programa irá pular todo o caso do else, sem ter de fazer novos testes. Em outros casos, pode ser que a expressão seja verdadeira em mais do que um teste, tal como exemplificado no Código 3.6:

Código 3.6 Testes não-exclusivos

```
if ((numero % 2) == 0)
   System.out.print(“o numero é par...”);
if (numero > 0)
   System.out.print(“o numero é positivo...”);
if ((numero >= 1000) && (numero <= 3000))
   System.out.print(“ 1000 < numero < 3000”);
```

No primeiro teste estamos usando o operador %, também chamado de mod. Esse operador calcula qual é o resto da divisão inteira do primeiro operador (nesse caso, `numero`) pelo segundo (nesse caso, 2). É comum usar-

mos o mod para verificar se um número é divisivel pelo outro, tal como neste exemplo: se o número for divisível por 2 significa que ele é par. No segundo teste estamos verificando se o número é maior do que zero. No terceiro teste queremos verificar se o número está entre 1000 e 3000. Esse é um caso de teste composto, uma vez que deve verificar se duas condições são verdadeiras ao mesmo tempo (operador E): número tem de ser maior ou igual do que 1000 e ao mesmo tempo menor ou igual do que 3000.

Observe que, neste exemplo, se o número for igual a 1200, todos os testes serão verdadeiros e, portanto, todos os comandos do if serão executados. Caso o número seja 999, apenas o segundo teste será verdadeiro. Caso o número seja 1001, o segundo e o terceiro serão verdadeiros. Assim, pode-se concluir que é uma sequência de ifs não-exclusiva.

Como pode ser visto na linha 5 do Código 3.6, alguns testes podem ser mais complexos e para tanto irão requerer expressões lógicas compostas. No Código 3.7 queremos resolver um problema típico de jogos 2D: precisamos saber se as coordenadas de um objeto recaem dentro ou fora da tela. As variáveis objetoX e objetoY contêm as coordenadas, em pixels, do objeto em questão. As variáveis alturaTela e larguraTela contêm o tamanho, em pixels, da tela do jogo.

Código 3.7 Objeto dentro ou fora da tela

```
if ((objetoX < larguraTela) && (objetoX >0)
   && (objetoY < alturaTela) && (objetoY > 0)
   // O objeto está dentro da tela
else
   // O objeto esta fora da tela
```

O if tem uma estrutura simples e, com um pouco de experiência e prática, você será capaz de criar expressões lógicas complexas. Não há muito mais a ser dito em relação ao if, apenas mais um pequeno detalhe: a expressão lógica pode ser complexa, como foi visto no Código 3.7. Mas, em última instância, o que importa é que haja um resultado lógico causado por essa expressão. Isso permite que utilizemos o próprio valor lógico no lugar de uma expressão, tal como exemplificado no Código 3.8.

Neste exemplo, a variável fim é usada para armazenar uma expressão lógica que verifica se o jogo deve terminar. No caso de `vidas` ser menor do que

zero ou a saúde ter chegado ao `fim`, a variável fim irá valer verdade. Neste caso, colocamos apenas a variável no lugar da expressão lógica.

Código 3.8 Expressão lógica simplificada

```
boolean fim;

fim = (vidas < 0) || (saude < 0);
if (fim)
   System.out.print("você perdeu!!!");
```

Como o controle de fluxo é muito usado e os programadores gostam de minimizar o número de teclas digitadas num programa, há algumas alternativas para simplificar seu uso e economizar o uso do teclado. Um desses atalhos é o operador ternário condicional. Dizemos que ele é ternário pelo fato de ser um operador com três operandos. Possui a seguinte sintaxe:

retorno = (expressão lógica)? (comando 1): (comando 2)

O operador condicional irá inicialmente executar a expressão lógica. Caso o resultado seja verdadeiro, executará o comando 1 e retornará o seu resultado, caso contrário, executará o comando 2, retornando o seu resultado. O Código 3.9 ilustra o seu uso: caso o número seja positivo, ele será incrementado, caso seja negativo, será decrementado. A variável inteira `resultado` irá receber esse novo valor de número.

Código 3.9 Operador ternário condicional

```
resultado = (numero > 0) ? (numero ++) : (numero --)
```

3.2. O comando Switch

Outra forma de simplificar a escrita do if consiste em tornar uma sequência encadeada de ifs mais enxuta e elegante, já que em algumas situações, pelo fato de haver muitas condições possíveis, podem ocorrer muitas concatenações de ifs, tornando o código um pouco (ou muito) intragável. Uma típica situação dessas é quando desejamos implementar menus e submenus: devemos ler uma opção e percorrer uma série de opções, para verificar qual foi a escolhida pelo usuário. O comando switch permitirá escrever essa sequência de forma mais simplificada. Observe a sua sintaxe no Código 3.10.

Código 3.10 Sintaxe do comando switch

```
1     switch (expressão) {
2        case valor1:
3           comandos 1
4           break;
5        case valor2:
6           comandos 2
7           break;
8           .
9           .   // (quantos casos quiser)
10          .
11       case valorN:
12          comandos N
13          break;
14       default:
15          comandos default
16    } // final do bloco do switch
```

A expressão, na linha 1, tem um significado ligeiramente diferente do que no if: essa expressão não deve retornar true / false, mas pode retornar valores que sejam destes tipos: inteiros, char, short, byte. É mais ou menos óbvio porque não é booleano: se tivéssemos apenas duas opções a serem tratadas não teríamos por que usar o switch, bastaria usar um if... Um detalhe importante: a expressão do switch não pode retornar o tipo string ou float. Após analisar o resultado da expressão, o comando switch irá percorrer cada um dos casos e verificar qual é o valor que lhe corresponde. Quando esse valor for encontrado, serão executados todos os comandos que estão dentro do bloco desse caso. Há um caso especial, chamado default, em que não se realiza nenhum teste de valor. A ideia aqui é que seu bloco (no exemplo, corresponde ao comando da linha 15) sempre será executado.

O comando break não é de exclusividade do switch. Sua função é a seguinte: faz com que o programa saia do bloco em que se encontra e continue a partir do bloco seguinte. No caso do switch, o comando break faz com que o fluxo do programa pule para o final do switch. O break é opcional ao usar o switch. Entretanto, se ele não for usado, após um caso executar seus comandos correspondentes, o switch continuará verificando se o resultado da expressão é igual a algum dos valores listados. Se o default estiver presente (mencionamos que ele é opcional, portanto, pode não existir), o programa irá executar os comandos do default, mesmo já tendo executado algo anteriormente. O Código 3.11 ilustra um exemplo do switch.

Código 3.11 Exemplo do comando switch

```
1  switch (vidas) {
2     case 0:
3        System.out.println("Cuidado, você não tem mais vidas…");
4        break;
5     case 1:
6        System.out.println("Você tem só mais uma vida…");
7        break;
8     case 2:
9     case 3:
10       System.out.println("Você tem algumas poucas vidas…");
11       break;
12    default:
13       System.out.println("Você ainda tem muitas algumas vidas…");
14 }
15 // continua fluxo do programa
```

No exemplo anterior, a variável `vidas` é do tipo inteira. Neste exemplo consideramos que um número de vidas entre 2 e 3 é pouco e mais do que isso é um valor suficiente para permanecer "tranquilo". Nesse caso a expressão é apenas a variável `vidas`, portanto, o resultado será o próprio valor contido na variável. Se não houver mais vidas (vidas = 0) então receberemos o comunicado de cuidado, conforme impresso na linha 3. Após essa mensagem, há um break, fazendo com que o programa salte para o final do escopo do switch, portanto linha 15. Caso o jogador tenha apenas uma vida restante, será avisado disso, pois o teste será verdadeiro para a linha 5. Caso contrário, o fluxo do switch continuará e irá testar se vidas é igual a 2. Observe que, nesse caso, o comando seguinte é um novo teste de caso. Essa construção é possível dentro do switch e fará com que mais do que um valor possa usufruir dos mesmos comandos. Então, se vidas for igual a 2 ou 3 será impressa a mensagem de que o jogador possui poucas vidas, e, assim que esse aviso for dado, o fluxo será desviado para o final do switch, em função do break da linha 11. Se o programa chegou até a linha 12 é porque não entrou em nenhum dos casos e portanto todos falharam. Isso quer dizer, no exemplo, que o número de vidas é maior do que 3, portanto será avisado ao jogador que ainda lhe restam bastantes vidas (mais do que 3)... Não é necessário colocar o break após o caso default, pois abaixo dele não há mais nada, fazendo com que após a execução dos seus comandos o programa saia do switch de qualquer forma.

Código 3.12 Comando switch usado para menus

```
int opcao;
System.out.println ("1 - Iniciar Jogo ");
System.out.println ("2 - Configurar jogo");
System.out.println ("3 - Creditos");
System.out.println ("4 - Sair do Jogo ");
System.out.println ("Entre com sua opção: ");
opcao = in.nextInt();

switch (opcao) {
   case 1:
      System.out.println("inicializando jogo…");
      InitJogo();  // método que irá iniciar o jogo
      break;
   case 2:
      ConfigJogo();  // método que chamará a tela de configuração
      break;
   case 3:
      ImprimeCreditos();  // método que irá imprimir os créditos
      break;
   case 4:
      sairJogo();    // método que termina o jogo
      break;
   default:
      System.out.println("Opção inválida…");
}
// continua fluxo do programa
```

O Código 3.12 apresenta uma situação comum de uso para o switch, que consiste na construção de menus.

O Código 3.12 irá inicialmente imprimir um menu (por favor, tenha um pouco de paciência, por enquanto está no modo texto, porém mais adiante, caso você persevere, será capaz de imprimir em modo gráfico...). Logo em seguida, será pedido para que o usuário entre com uma opção, colocando-se a leitura do mesmo na variável `opcao`. O comando switch será usado para verificar que opção foi digitada. Neste exemplo, a expressão também é simples, resumindo-se à própria avaliação da variável `opcao`. Outra novidade que podemos observar neste exemplo é que os comandos referentes a um caso podem ser compostos de várias linhas (aqui cada caso possui dois comandos: um método que tratará da situação específica e uma impressão de mensagem). Diferentemente de outros casos de comandos compostos, aqui não é necessário criar um bloco com os símbolos "{" e "}". Isso se deve ao fato de que os casos não são exatamente blocos, uma vez que, ao terminar um deles, é possível continuar percorrendo o fluxo do switch, caso não haja um break avisando para pular para o final.

3.3. Um pouco mais sobre controle de fluxo

Já dissemos muitas vezes que o computador sabe fazer poucas coisas e que, se é que algum dia as máquinas irão adquirir uma inteligência superior aos humanos, esse dia ainda está muito longe. Entretanto, há algo em que o computador nos supera de longe: ele é capaz de efetuar operações matemáticas a uma velocidade absurdamente maior do que nosso cérebro. Enquanto os humanos são capazes de demorar alguns segundos para efetuar uma soma de dois números grandes, o computador irá levar uma fração de microssegundos. Assim, há uma coisa que o computador é capaz de fazer e nós não: efetuar um volume muito grande de contas e operações. Um jogo é composto por várias imagens estáticas, que ao serem mostradas a uma taxa de 30 imagens diferentes por segundo, teremos a sensação de uma sequência em movimento. Suponha que a resolução da tela seja de 1024 x 768, o que não é das maiores. Nesse caso, uma imagem será composta por 786.432 pixels, o que significa que em um segundo serão calculadas a cor de mais de 23 milhões de pixels.

De fato, os computadores ganham grande utilizade em nosso dia a dia pelo fato de serem capazes de resolver muitas contas em pouco tempo.

Até agora, os programas desenvolvidos são muito simples e contêm poucas operações. Com os conceitos apresentados até o momento, você precisa escrever praticamente uma sequência linear dos comandos a serem executados, com algumas pequenas bifurcações, provenientes do if. No capítulo seguinte, será apresentado o conceito de um laço, que é um mecanismo muito poderoso de controle de fluxo, permitindo que um escopo possa ser executado um número muito grande de vezes. Após a introdução desse conceito, você será capaz de desenvolver programas muito mais interessantes e divertidos dos que vêm sendo mostrados até o momento...

EXERCÍCIOS

E3.1. Escreva um programa que leia a sua data de aniversário e imprima qual é o seu signo.

E3.2. Usando if, implemente: se x é igual a 1 ou 2, imprima “1 ou 2”, se for igual a 3 ou 4, imprima “3 ou 4”.

E3.3. Reescreva o E3.2, usando switch em vez de if.

E3.4. Explique por que não faz muito sentido que a expressão do comando switch permita o retorno de um resultado do tipo real (float / double).

E3.5. O índice de massa corporal (IMC) de uma pessoa é calculado dividindo-se o peso pela sua altura ao quadrado. Quando esse valor está abaixo de 19, dizemos que a pessoa está magra. Se o valor estiver entre 20 e 25, dizemos que a pessoa está com o peso ideal. Entre 26 e 30 dizemos que a pessoa está acima do peso, e, caso o valor ultrapasse 31, dizemos que a pessoa está obesa, sendo a situação prejudicial para sua saúde. Faça um programa que leia o peso e a altura de uma pessoa e imprima em que situação a pessoa se encontra. (Atenção: para uma verificação correta há outros fatores envolvidos, tais como sexo, idade e biótipo, portanto, esse teste é apenas para fins de exercício, não devendo ser usado como um teste de verdade.)

E3.6. Escreva um programa que leia o vértice superior esquerdo e inferior direito que descreve um retângulo no plano e logo em seguida leia a coordenada de um ponto no espaço. O programa em seguida deve responder se o ponto está dentro ou fora do retângulo. Assuma que o retângulo está alinhado com os eixos cartesianos.

E3.7. Estenda o Exercício E3.6 para o espaço 3D, sendo que os vértices neste caso devem ser compostos por três coordenadas: leia os dois vértices que descrevem os dois extremos do cubo e em seguida leia um ponto no espaço 3D e verifique se o ponto está dentro ou fora do cubo.

E3.8. Repita os Exercícios E3.6 e E3.7 para uma circunferência e uma esfera: leia as coordenadas do centro e o seu raio e em seguida as coordenadas de um ponto, para depois responder se o ponto está fora ou dentro da circunferência/esfera.

E3.9. Faça um programa que leia dois pontos que definem uma reta no plano. Em seguida, o programa deverá ler as coordenadas do centro e o raio de uma circunferência. Finalmente, o programa deverá calcular se a reta tem interseção ou não com a circunferência. Em caso positivo, o programa deverá imprimir quais são as coordenadas de interseção.

E3.10. Repita o Exercício E2.5 do capítulo anterior, porém usando o comando switch.

E3.11. Faça um programa que leia três números e imprima qual é o menor, qual o valor do meio e qual o maior.

E3.12. Repita o exercício anterior, porém desta vez tratando o caso de os números serem repetidos dois a dois. O programa deverá nestes casos dizer que dois são repetidos e ambos os maiores ou menores. No caso de os três números serem repetidos, o programa deverá avisar que todos são iguais.

E3.13. Faça um programa que verifica se uma circunferência está totalmente dentro, parcialmente dentro ou totalmente fora de um retângulo. Para ler os dados da circunferência, o programa deverá pedir os valores do centro e do raio. Para ler os dados do retângulo, o programa deverá pedir os valores das coordenadas superior esquerda e inferior direita.

E3.14. Escreva um programa que leia três coordenadas de vértice de um triângulo e verifique se é realmente um triângulo (para que três vértices componham um triângulo, é necessário que não estejam alinhados numa mesma reta).

E3.15. Escreva um programa que leia o canto superior esquerdo e inferior direito de um retângulo e verifica se é um quadrado (todos os lados são iguais).

E3.16. Escreva um programa que leia as coordenadas superiores esquerda e inferiores direita de dois retângulos e verifica se ambos estão colidindo ou não. Dizemos que dois retângulos colidem entre si quando pelo menos um dos vértices de um dos retângulos está dentro do outro retângulo. (Obs: esse cálculo corresponde ao típico cálculo de colisão em jogos 2D, em que os elementos de uma cena são descritos por *sprites*, como será visto em capítulos posteriores.)

E3.17. Estenda o problema anterior para calcular se houve colisão entre duas caixas 3D. (Obs: esse cálculo corresponderá ao cálculo de colisão em jogos 3D, em que serão criadas caixas envolventes para os objetos geométricos de uma cena.)

E3.18. Escreva um programa que leia os dados de duas circunferências (centro e raio) e verifique se as duas estão se chocando ou não. Justifique por que esse método é computacionalmente mais eficiente do que o cálculo de colisão entre retângulos.

E3.19. Repita o Exercício E3.18 para esferas. Sabendo que podemos criar esferas que envolvam objetos geométricos de uma cena e usá-las para o cálculo de colisão entre os elementos. Justifique por que essa pode vir a ser uma estratégia menos precisa que usar caixas envolventes.

Capítulo

4
Métodos e Soluções

Métodos implementam soluções de problemas específicos que chamamos de algoritmos. Neste capítulo, apresentamos o conceito de método, as suas características e os cuidados no seu uso. Os métodos começam a ficar interessantes quando envolvem repetições e autorreferências. Neste livro apresentamos essas situações como sendo soluções iterativas ou recursivas.

4.1. Métodos

Java é uma linguagem orientada a objetos que usa os conceitos de classe, objeto e método. Entretanto, nesta primeira parte do livro, não estaremos usando Java em sua plenitude, isto é: não definiremos classes nem criaremos objetos a partir delas. No momento basta sabermos que uma classe contém métodos que realizam tarefas. O método `main` é como qualquer outro método, exceto que ele é diretamente chamado pela máquina virtual Java. Um método pode chamar outros métodos. Por exemplo, o método `main` pode chamar o método `System.out.print` predefinido no Java, como vimos nos capítulos iniciais deste livro. O método `main` também pode chamar um método que você mesmo definiu.

Inicialmente, você pode imaginar um método como sendo uma função, conforme aprendeu em matemática, *i.e.*: uma função mapeia valores de seu

domínio em valores de seu contradomínio. Por exemplo, a função $f(x,y)= x^2 + y^2$ mapeia (1,2) a 5 e (2,3) a 13, ou seja: $f(1,2)= 5$ e $f(2,3)= 13$. Funções são a forma mais pura de um modelo de computação. Uma função matemática lembra uma máquina que recebe valores para os seus argumentos de entrada, processa-os e retorna um valor como saída. Nessa comparação, você primeiro define a máquina de forma geral ($f(x, y)= x^2 + y^2$) e depois a usa para diferentes valores de seus argumentos de entrada (*e.g.* para x=1 e y=2, a máquina expele o número 5). Métodos implementam funções matemáticas, que funcionam como máquinas (Figura 4.1a). Podemos generalizar ainda mais esse conceito de método, aceitando argumentos de entrada de tipos diferentes e permitindo que as máquinas produzam "efeitos colaterais", tais como escrever uma mensagem ou criar um arquivo. Máquinas com efeitos colaterais deixam de ser equivalentes a funções matemáticas. Neste caso, podemos dizer que as máquinas deixam de ser **funcionais** e passam a ser **procedimentais**, no sentido de que seguem um procedimento, claro, sequencial e com múltiplos efeitos. Figura 4.1.b ilustra essa máquina genérica. Métodos em Java implementam todos esses tipos de máquinas. A linguagem Java também nos permite definir um método sem implementá-lo; por exemplo: podemos definir as características de uma máquina (dando um nome e especificando os tipos dos parâmetros de entrada e o tipo de saída) sem escrever e detalhar como ela realiza tarefas. Nesse caso, os métodos são chamados de **métodos abstratos** (***abstract***). Há outros tipos de métodos que só mencionaremos mais adiante neste livro. Nesta primeira parte, queremos apenas tratar de métodos que implementam funções e tarefas de maneira independente e genérica (sem a necessidade da criação prévia de objetos) – típicas de um modelo de computação genérica. Chamamos tais métodos de **métodos estáticos** (***static***).

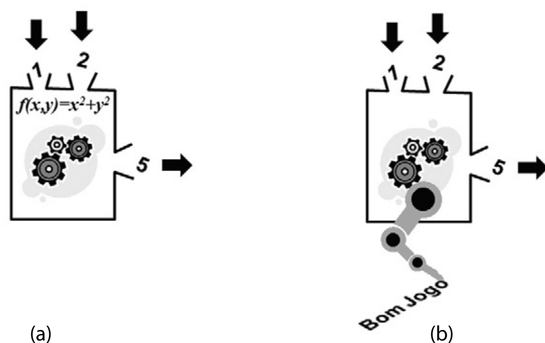


Figura 4.1 (a) Função como uma máquina que recebe valores para seus parâmetros de entrada e retorna um valor como saída. (b) Máquina mais geral que também produz efeitos colaterais (tal como escrever a mensagem "Bom Jogo").

Neste momento de sua leitura, cabe uma observação muito importante. Aconselhamos que você sempre projete suas máquinas (ou melhor dizendo: seus métodos) da maneira mais genérica, limpa e concisa possível. Métodos implementam tarefas, e você deve sempre tentar defini-las como partes menores de tarefas maiores, de maneira que métodos chamem outros métodos auxiliares genéricos. No caso de funções matemáticas, por exemplo, o problema de calcular a série 1 + 1/1! + 1/2! + … é mais facilmente resolvido quando primeiro definimos um método auxiliar mais simples e genérico, que calcula o fatorial de um número qualquer, isto é: *n*!. Lendo este livro, mantenha essa **subdivisão de tarefas genéricas** sempre em mente – habitue-se a decompor problemas em subproblemas.



Nota

Algumas linguagens de programação, chamadas **Linguagens Funcionais**, como Lisp e Haskell, consideram o ato de computar como sendo o ato de avaliar funções matemáticas, sem que o estado do mundo (*e.g.* variáveis) seja alterado. Lisp, por exemplo, é uma linguagem que pode ser entendida como sendo simplesmente expressões dentro de expressões que sempre retornam valor. Em oposição, temos linguagens de programação denominadas **Linguagens Procedimentais** (tais como C, C++ e Java), que definem procedimentos, isto é: comandos imperativos em uma ordem específica que podem alterar o estado do mundo (*e.g.* mudanças de valores de variáveis ou efeitos colaterais de uma maneira geral). Porém, todas as linguagens permitem uma fuga para o estilo que lhe é oposto por natureza. Por exemplo, podemos escrever procedimentos em Lisp e funções em Java. Retornaremos a essas questões quando falarmos em recursão neste livro.

4.2. Criando um método estático

Até o capítulo anterior, você usou métodos já disponíveis na linguagem Java. Agora vamos ver como criar seus próprios métodos.

A sintaxe de um método é a seguinte:

```
modificadores tipoRetorno nome (argumentos)
{
    comandos
}
```

onde a primeira linha é chamada de **assinatura** do método (*method signature*) ou cabeçalho do método (*method header*) e o conjunto de *comandos* é denominado corpo do método. O exemplo da Figura 4.2 ajuda a explicar

os elementos de um método. Nesse exemplo, o método tem o nome de `max3`, recebe três números inteiros como argumentos (`a`, `b`, `c`) e retorna o maior deles. Analise o código e veja que o valor da expressão a retornar (no caso, simplesmente o valor da variável `max`) é compatível com o tipo de retorno especificado na assinatura do método (*i.e.*: `int`). Os modificadores estabelecem a natureza do método, isto é, a sua visibilidade e a forma como pode ser utilizado por outras partes do programa Java. No caso da Figura 4.2, temos que o método é público (`public`), no sentido de que qualquer parte do programa pode usá-lo, e o método é estático (`static`), significando uma tarefa independente e genérica (sem a necessidade da criação prévia de um objeto).

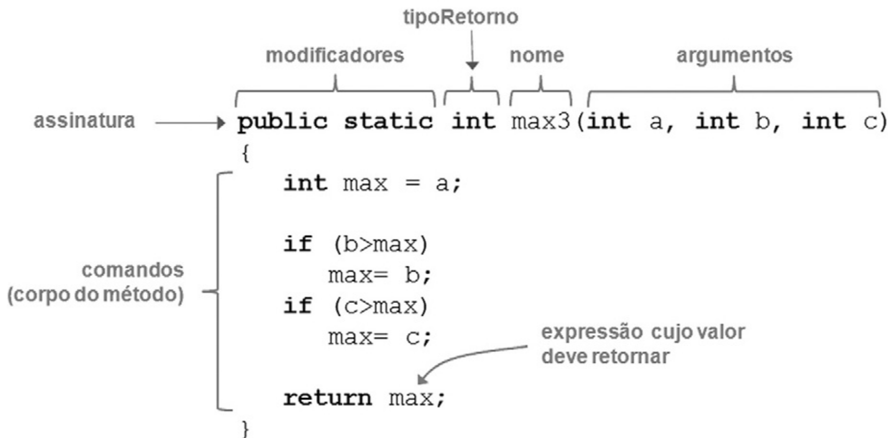


Figura 4.2 Elementos de um método.

O Código 4.1 mostra como definir e usar um método estático. A mais importante observação, no momento, é que passamos "valores" para um método e não as variáveis em si. Note que os nomes das variáveis no `main` não precisam ser os mesmos usados pelo universo de um método (porque são valores passados). Se a variável `a` dentro do método mudasse de valor, `a` continuaria inalterada no `main` após a chamada do método, pois, apesar de terem o mesmo nome, a variável `a` no método `max3` é uma cópia da variável `a` do método `main`. Voltaremos a essa questão de passagem de valores no final da Seção 4.4.1 e no Capítulo 5.

Código 4.1 Definição e uso do método max3

```
public class Cap04Ap01
{
   public static int max3(int a, int a2, int a3)
   {
      int max = a;

      if (a2>max)
         max= a2;
      if (a3>max)
         max= a3;

      return max;
   }

   public static void main(String[] args)
   {
      int a=1;
      int b=7;
      int c=10;
      int i = max3(a,b,c);
      System.out.println("Máximo= "+i);
   }
}
```

Um método pode não retornar qualquer valor e, nesse caso, a assinatura do método deve ter `void` como *tipoRetorno*. Geralmente, esses métodos são usados pelos seus efeitos colaterais. O Código 4.2 apresenta um método que apenas imprime a análise do discriminante da solução de uma equação do 2º grau.

Código 4.2 Exemplo de método que não retorna valor. O discriminante $\Delta=b^2-4ac$ da equação $ax^2+bx+c=0$ tem a seguinte análise: sem raízes reais se $\Delta < 0$, raízes reais iguais se $\Delta=0$ e raízes reais diferentes se $\Delta \geq 0$.

```
public class Cap04Ap02
{
   public static void discriminante(int a, int b, int c)
   {
      double d= b*b-4*a*c;
      if (d<0)
      {
         System.out.println("Não existem raízes reais");
      }else
      {
         if (d==0)
         {
            System.out.println("Existem duas raízes reais iguais");
         }else
         {
            System.out.println("Existem duas raízes reais diferentes");
         }
      }
   }

   public static void main(String[] args)
   {
```

Código 4.2 Exemplo de método que não retorna valor. O discriminante $\Delta=b^2-4ac$ da equação $ax^2+bx+c=0$ tem a seguinte análise: sem raízes reais se $\Delta < 0$, raízes reais iguais se $\Delta=0$ e raízes reais diferentes se $\Delta \geq 0$. (cont.)

```
      int a=1;
      int b=6;
      int c=9;
      discriminante(a,b,c);
   }
}
```

4.3. Sobrecarga de métodos

Código 4.3 Sobrecarga de métodos

```
public class Cap04Ap03
{
   public static int min(int a, int b)           // metodo 1
   {
      if (a<b)
         return a;
      return b;
   }

   public static double min(double a, double b)  // metodo 2
   {
      if (a<b)
         return a;
      return b;
   }

   public static int min(int a, int b, int c)    // metodo 3
   {
      return min(min(a,b),c);
   }

   public static void main(String[] args)
   {
      System.out.println(min(7,2));
      System.out.println(min(5,3,1));
      System.out.println(min(7.0,2.0));
   }
}
```

Métodos podem ter o mesmo nome e lista de argumentos diferentes. Esse recurso é denominado sobrecarga de método (*method overloading*). No exemplo do Código 4.3, o compilador tem condições de decidir que a chamada `min(7,2)` será resolvida pelo método 1, `min(5,3,1)` pelo método 3 e `min(7.0,2.0)` pelo método 2. A impressão dos resultados do Código 4.3 é a seguinte:

```
2
1
2.0
```

Sobrecarregar métodos é uma boa prática de programação quando encontramos tarefas que realizam essencialmente a mesma ação apesar de terem número e tipos de argumentos diferentes. A sobrecarga de métodos é exclusivamente baseada na lista de argumentos. Mudanças nos *modificadores* ou *tipoRetorno* não sobrecarregam métodos.

O compilador sempre procura pelo método mais específico, num processo denominado **resolução de sobrecarga** (***overload resolution***). Por exemplo, no Código 4.3, se a chamada fosse `min(7,2.1)` o compilador entenderia que o método 2 é o mais específico, após a conversão automática do valor 7. Entretanto, algumas situações podem deixar o compilador sem condições de encontrar o método mais específico, tal como a chamada `min(3,7)` e as seguintes assinaturas:

```
public static double min(int a, double b)
...
public static double min(double a, int b)
```

Nesse caso o compilador exibiria uma mensagem de erro dizendo que a referência a `min` é ambígua.

4.4. Soluções iterativas com laços

O que você faria se tivesse de imprimir 50 vezes o *string* "Bom Jogo!"? Com o que você aprendeu nos capítulos anteriores, seria necessário escrever 50 vezes o comando de impressão – algo nada prático! E se você quiser fazer um programa que pede ao usuário quantas vezes ele gostaria de imprimir o *string* "Bom Jogo!"? Nesse caso, você talvez nem saiba sugerir uma solução, mesmo que tediosa como a anterior. Com o que aprendeu até esta seção do livro, você simplesmente não sabe o que fazer. Você necessita de novas capacidades de programação!

As linguagens de programação oferecem uma estrutura de controle de fluxo chamada "laço" (ou "*loop*", em inglês) que controla o número de vezes que uma sequência de comandos é repetida. A parte mais importante dessa estrutura é a condição de continuação do laço que, neste livro, chamamos de "*teste-de-continuação*". Esse teste é uma expressão Booleana que, se for verdadeira, permite a execução da sequência de comandos do laço. Cada execução da sequência de comandos é chamada de "uma iteração do laço". Na maioria das linguagens, os controles de laço mais usados são: `while`, `for` e `do-while`.

4.4.1. O Laço `while`

A sintaxe do laço `while` é a seguinte:

```
while (teste-de-continuação)
{
   comandos
}
```

O primeiro problema mencionado no início da Seção 4.4 pode ser solucionado com o seguinte laço que imprime 50 vezes o texto “Bom Jogo!”:

```
int n = 0;
while (n < 50)
{
   System.out.println("Bom Jogo!");
   n++;
}
System.out.println("Fim das mensagens");
```

onde os comandos `System.out.println("Bom Jogo!")` e `n++` são executados até que `n < 50` seja falso. Nesse caso, o laço termina e o próximo comando depois do laço é executado. Uma outra variante dessa solução seria:

```
int n = 50;
while (n > 0)
{
   System.out.println("Bom Jogo!");
   n--;
}
System.out.println("Fim das mensagens");
```

A solução para o segundo problema mencionado na introdução da Seção 4.4 (aquele no qual o usuário estabelece o número de impressões) pode ser a apresentada no Código 4.4, cuja saída está na Figura 4.3.

Um problema mais interessante é o cálculo do fatorial implementado como um método. A ideia, neste caso, é usar uma variável auxiliar `f` que vai acumulando sucessivos produtos para obter $n! = n \times (n\text{-}1) \times (n\text{-}2) \times \ldots \times 1$. Uma possível solução é a seguinte:

```
public static int fat(int n)
{
   int f;
   f = 1;
   while (n>1)
   {
      f *= n; // equivalente a f = f * n;
      n--; // equivalente a n = n-1
   }
   return f;
}
```

Esse código pode ser ainda mais conciso:

```
public static int fat(int n)
{
   int f;
   f = 1;
   while (n>1)
      f *= n--;
      return f;
}
```

Código 4.4 Laço de while

```
import java.util.Scanner;

public class Cap04Ap04
{
   public static void main(String[] args)
   {
      Scanner entrada = new Scanner(System.in);
      System.out.print("Entre numero de linhas a imprimir: ");
      int n = entrada.nextInt();
      while (n > 0)
      {
         System.out.println("Bom Jogo!");
         n--;
      }
      System.out.println("Fim das mensagens");
   }
}
```

```
Output - Cap04Ap03 (run)
run:
Entre numero de linhas a imprimir: 3
Bom Jogo!
Bom Jogo!
Bom Jogo!
Fim das mensagens
BUILD SUCCESSFUL (total time: 15 seconds)
```

Figura 4.3 Saída do programa no qual o usuário determina o número de linhas impressas (Código 4.4).

A esta altura, você pode achar interessante acompanhar a evolução das variáveis dentro de um laço. No caso do exemplo de fatorial, temos para n = 4:

f	n	
1	4	valores na entrada do laço
4	3	1ª iteração permitida (pois n=4 > 1): f = 1 x 4 e n se torna 3
12	2	2ª iteração permitida (pois n=3 > 1): f = 4 x 3 e n se torna 2

24	1	3ª iteração permitida (pois n=2 > 1): f = 12 x 2 e n se torna 1
24		laço termina (pois n=1) e o valor de f = 24 retorna

aqui cabe uma observação importante: não adquira o mau hábito de só programar acompanhando a evolução de valores das variáveis de um método. Apresentamos o acompanhamento de f e n apenas para você compreender o processo.

O Código 4.5 apresenta um programa completo para o cálculo do fatorial de um número *n*. Temos aqui novamente a oportunidade de ver que o que passa para o método `fat` é o valor da variável `n`. De fato, dentro do método `fat`, a variável `n` termina com o valor 1, porém a variável `n` no método `main` continua com o mesmo valor.

Código 4.5 Fatorial como int

```
import java.util.Scanner;
public class Cap04Ap05
{
   public static int fat(int n)
   {
      int f;
      f=1;
      while (n>1)
      {
         f *= n;
         n--;
      }
      return f;
   }

   public static void main(String[] args)
   {
      Scanner entrada = new Scanner(System.in);
      System.out.print(“Entre numero: ”);
      int n = entrada.nextInt();
      System.out.println(“Fatorial de ”+n+” = ”+fat(n));
   }
}
```

4.4.2. Erros Numéricos em Soluções Iterativas

Implemente o programa do Código 4.5 e tente obter os seguintes valores de fatorial:

11! = 39916800

12! = 479001600

13! = 6227020800

Por que você não conseguiu obter 13!? A resposta é que a faixa de valores para o tipo `int` é -2^{31} a $(2^{31}-1)$, ou seja: -2147483648 a 2147483647, visto que o tamanho de armazenagem é 32 bits com sinal. Para irmos além de 12!, devemos usar o tipo `long` de 64 bits com sinal, que permite a faixa de -2^{63} a $(2^{63}-1)$, isto é: -9223372036854775808 a 9223372036854775807. Primeiro altere o Código 4.5 para ter a assinatura `public static long fat(int n)` e a declaração de `f` como `long f;`. Depois tente obter os seguintes valores:

17! = 355687428096000 (15 dígitos)

20! = 2432902008176640000 (19 dígitos)

21! = 51090942171709440000 (20 dígitos)

Você deve ter encontrado um valor totalmente errado para 21!. Isso ocorreu porque a operação `f= f*n` ultrapassou o limite superior da faixa do tipo `long`. Somente o uso de uma classe de números chamada de `BigInteger` pode lidar com valores inteiros de qualquer tamanho (veja Nota no final desta seção).

Erros numéricos são sutis e traiçoeiros. Algumas vezes esses erros não afetam o resultado, mas podem representar desperdício de tempo valioso de processamento. Um exemplo é o cálculo do número de Napier pela série: e = 1 + 1/1! + 1/2! + 1/3! + …≈ 2.71828182845904523536… que pode ser feito pelo seguinte método:

```
public static double napier(int n)
{
   double e = 0.0;
   while (n >= 0)
   {
      e += 1.0/fat(n);
      n--;
   }
   return e;
}
```

Supondo que `fat` retorna um valor tipo `long`, qualquer $n \geq 17$ contribui com uma parcela $1/n!$ da ordem de 10^{-15}, ou seja: nada contribui para a precisão de uma variável `double`, que só consegue representar 15 algarismos significativos aproximadamente. Além do mais, para $n > 20$, a parcela $1/n!$ produzirá números errados, mas que não afetam o resultado (gastando apenas tempo de processamento). Você pode implementar o método `napier` anterior e ver que para qualquer valor de n o valor se estabiliza em 2.718281828459045 (*i.e.*: 15 algarismos significativos após o ponto decimal).

Outros tipos de erros numéricos são muito difíceis de serem percebidos em laços. O acúmulo de erros em operações insuspeitas pode causar grandes estragos, silenciosamente. Nessa rota estão alguns números decimais que não são frações decimais infinitas, mas cuja representação binária é uma fração binária infinita. Por exemplo, a representação binária do número 0.1 é uma fração infinita: $0.1_{10} = 0.0\mathbf{0011}0011\mathbf{0011}0011\ldots_2$. Algumas operações com determinados números potencializam essa imprecisão. Por exemplo, $12*0.1 \neq 1.2$. Você pode testar esse fato com o seguinte código que detectará a diferença:

```
if (1.2==12*0.1)
   System.out.println("igual");
else
System.out.println("diferente");
```

Ou simplesmente execute `System.out.println(12*0.1);` que imprimirá 1.200000000000000**2**. O seguinte laço também revela esse tipo de perigo:

```
double soma=0.0;
int n=10;
while (n>0)
{
   soma += 0.1;
   n--;
}
System.out.println(soma);
```

que imprimirá soma como 0.9999999999999999.

Por segurança, em casos críticos, recomendamos o uso de comparações dentro de uma tolerância preestabelecida (*e.g.*: `public static final double TOL = 1.0e-5;`):

```
if (Math.abs(a-b)<TOL) {...}
```

A falha de um míssil americano Patriot na interceptação de um míssil inimigo Scud, durante a Guerra do Golfo em 1991, foi uma notícia amplamente divulgada na Internet que creditava a falha a um erro com frações binárias infinitas. Você deve estar muito atento a esse tipo de erro, pois, em um jogo, frações binárias infinitas podem provocar comportamentos errôneos nas simulações físicas.

Outros erros numéricos são oriundos de uma má previsão da faixa de valores das grandezas pertinentes ao problema em questão. Quando a eco-

nomia de espaço e tempo é crucial em aplicações de alto desempenho ou em hardware limitado (*e.g.* celulares ou sistemas embarcados), costumamos transformar valores cinemáticos decimais em valores inteiros. Por exemplo, valores da velocidade de um objeto detectada como um `double` de 64 bits podem ser convertidos para um inteiro de 16 bits com sinal. Nesses casos, o perigo está no fato de que alguns desses valores podem não ser representados corretamente. A explosão do foguete Ariane 5 em 1996 foi creditada a uma conversão desse tipo.



Nota

`BigInteger` é uma classe de números disponível no pacote `java.math.BigInteger`. A manipulação de números grandes requer operadores e métodos especiais. Esses números podem ser criados através do método `valueOf` ou do operador `new`. Esse assunto está além do escopo do presente capítulo, mas não deixa de ser instrutivo analisar o programa que permite calcular fatoriais sem limites. Implemente o programa a seguir e calcule 25! = 15511210043330985984000000. As diferenças em relação ao Código 4.5 estão marcadas em negrito.

```
import java.util.Scanner;
import java.math.BigInteger;
public class Fatorial
{
   public static BigInteger fat(int n)
   {
      BigInteger f;
      f= new BigInteger("1"); // ou: f = BigInteger.valueOf(1);
      while (n>1)
      {
         f = f.multiply(BigInteger.valueOf(n)); // requer metodo
         // especial para multiplicar
         n--;
      }
      return f;
   }
   public static void main(String[] args)
   {
      Scanner entrada = new Scanner(System.in);
      System.out.print("Entre numero: ");
      int n = entrada.nextInt();
      System.out.println(n+"! = "+fat(n));
   }
}
```

4.4.3. Os Laços `for` e `do-while`

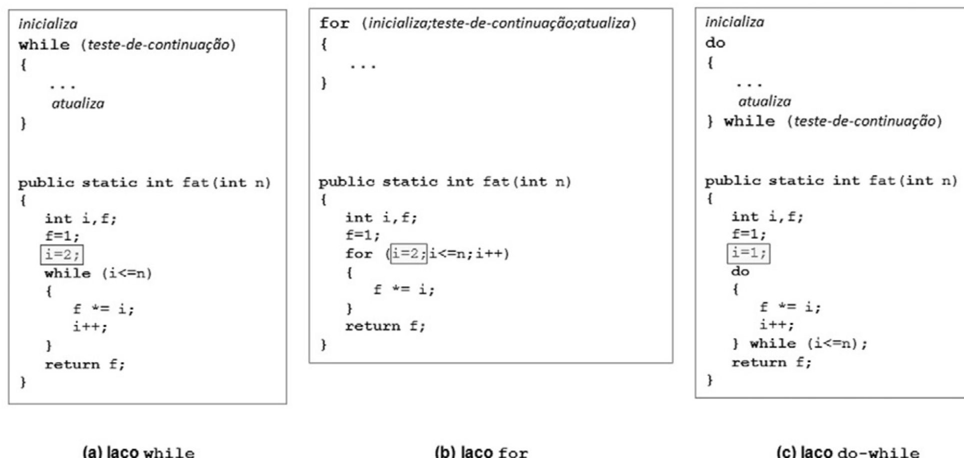


Figura 4.4 Sintaxe detalhada dos três principais tipos de laços junto com implementações equivalentes do método que calcula o fatorial.

Os laços `while`, `for` e `do-while` fazem essencialmente a mesma coisa. A maior diferença está no `do-while`, que permite uma iteração antes do "*teste-de-continuação*". A Figura 4.4 apresenta a sintaxe mais detalhada de cada um desses laços e a implementação equivalente do método que calcula o fatorial. Em todos os tipos de laço há os mesmos três elementos básicos:

- inicialização das variáveis (*inicializa*)
- condição de continuação do laço (*teste-de-continuação*)
- atualização das variáveis (*atualiza*)

Entre o `while` e o `for` a equivalência é total. Entretanto, para mantermos a equivalência no `do-while`, precisamos ajustar um pouco a lógica do método, visto que a primeira iteração é sempre realizada. O pequeno retângulo em destaque na Figura 4.4.c chama nossa atenção para o fato de que a variável `i` é inicializada com valor diferente no `do-while`. Ademais, uma análise mais cuidadosa do código da Figura 4.4.c revela que a primeira iteração faz uma conta que os outros laços evitam (*i.e.*: `f = f * i` para `f = 1` e `i = 1`).

Código 4.6 Jogo de adivinhação usando do-while

```
import java.util.Scanner;

public class Cap04Ap06
{
   public static void main(String[] args)
   {
      int segredo, chute, conta=0, a=1, b=6;
```

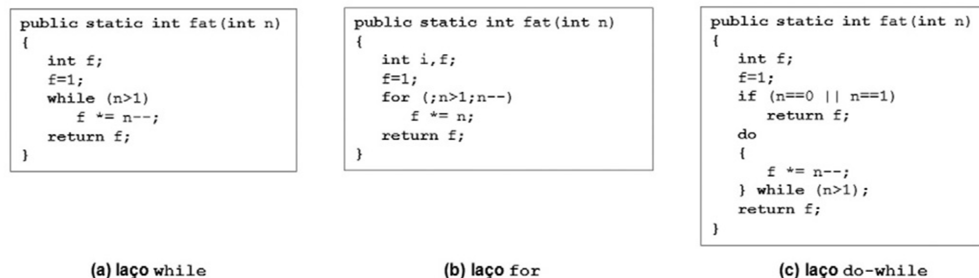
Código 4.6 Jogo de adivinhação usando do-while (cont.)

```
                Scanner entrada = new Scanner(System.in);
                segredo = a + (int)(Math.random()*b);  // numero no
                   intervalo [a,a+b)
                do
                {
                   System.out.print("Chute um numero: ");
                   chute = entrada.nextInt();
                   conta++;
                   if (chute==segredo)
                   {
                      System.out.println("ACERTOU! :-)");
                      conta = 0;
                   }
                   else
                   {
                      System.out.println("errou  :-(");
                   }
                } while (conta != 0);
             }
          }
```

O exemplo da Figura 4.4 pode nos levar à conclusão de que não há vantagem em usar o `do-while`. Porém, essa conclusão é falsa, pois o uso do `do-while` pode gerar códigos mais simples e enxutos em alguns casos. No exemplo do jogo de adivinhação do Código 4.6, o `do-while` evita a repetição de comandos para o primeiro "chute" do usuário. Neste exemplo usamos o método `random` da classe `Math`, que retorna um número no intervalo [0.0,1.0), isto é: um número entre 0.0 inclusive e 1.0 exclusive. A "fórmula" geral para se obter um número inteiro no intervalo [`a, a+b`) é a seguinte:

```
a + (int)(Math.random()*b)
```

No exemplo do código, a fórmula `1+(int)(Math.random()*6)` gera um número inteiro entre [1,6]. Devemos notar que `a` e `b` na fórmula não são os extremos do intervalo; por exemplo, `2+(int)(Math.random()*6)` gera um número inteiro no intervalo [2,7].



```
public static int fat(int n)
{
   int f;
   f=1;
   while (n>1)
      f *= n--;
   return f;
}
```

(a) laço `while`

```
public static int fat(int n)
{
   int i,f;
   f=1;
   for (;n>1;n--)
      f *= n;
   return f;
}
```

(b) laço `for`

```
public static int fat(int n)
{
   int f;
   f=1;
   if (n==0 || n==1)
      return f;
   do
   {
      f *= n--;
   } while (n>1);
   return f;
}
```

(c) laço `do-while`

Figura 4.5 Versões mais eficientes do método que calcula fatorial, usando `while`, `for` e `do-while`.

O exemplo da Figura 4.4 é meramente didático, posto que o método fatorial pode ser implementado de maneiras mais concisas e eficientes (como as apresentadas na Figura 4.5). A Figura 4.5 também serve para ilustrar que os componentes de um `for` não precisam estar todos definidos; podemos até mesmo usar `for(;;) {…}` para implementar um laço infinito (*i.e.* aquele que nunca para de executar iterações). O `for` também permite mais de uma variável na inicialização e na atualização, como no seguinte exemplo:

`for (i=0,j=0;`*teste-de-continuação*`;i++,j++)`

4.5. Soluções recursivas

Na Seção 4.4 vimos uma maneira eficiente de resolver problemas (*i.e.* de escrever algoritmos) através de repetições implementadas por construtores de laço (`while`, `for` e `do-while`). Veja novamente a implementação do fatorial na Figura 4.5a. Esses construtores de laço descrevem **processos iterativos**, em que a cada passo do processo podemos acompanhar a atualização dos valores das variáveis (`n` e `f` na Figura 4.5a). Os valores dessas variáveis descrevem o estado do mundo e, por isso, são denominadas **variáveis de estado**. Neste caso, podemos até interromper o processo, guardar os valores das variáveis e reiniciar o processo a partir desses valores em um outro momento. Podemos também imprimir os valores em cada passo, obtendo assim um "retrato" do estado do mundo em qualquer instante. Entretanto, essa não é a única forma de implementar algoritmos. Podemos também resolver problemas através de procedimentos que chamam a si próprios e que descrevem o que denominamos **processo recursivo**. No processo recursivo não temos variáveis que descrevem o estado do mundo. Os processos recursivos são geralmente muito mais simples e claros do que os processos iterativos.

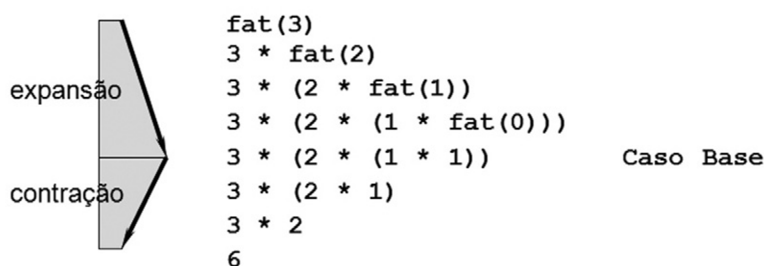


Figura 4.6 Processo recursivo com expansão e contração de uma cadeia de operações pendentes, que são guardadas em uma pilha na memória do computador. A contração começa quando o caso base é alcançado.

Considere a seguinte definição recursiva de fatorial, em que a função *fat* é definida em termos dela mesma:

$$fat(n) = \begin{cases} 1, \text{ se } n=0 \\ n \times fat(n-1), \text{ se } n>0 \end{cases}$$

Por exemplo: $fat(3) = 3 \times fat(2) = 3 \times (2 \times fat(1) = 3 \times (2 \times (1 \times fat(0))) = 3 \times (2 \times (1 \times 1) = 6$. Observe que a resolução é protelada até que um caso simples é alcançado ($fat(0) = 1$). O método recursivo que implementa essa função pode ser o seguinte:

```
public static long fat(int n)
{
   if (n==0)
      return 1;
   else
      return n*fat(n-1);
}
```

O processo de computar essa função é caracterizado por uma **cadeia de operações pendentes** que se expande e depois se contrai. Internamente, no computador, as operações são guardadas em uma pilha de informações na memória. A Figura 4.6 ilustra esse processo. Note que não identificamos variáveis de estado no processo recursivo.

Escrever algoritmos recursivos requer o desenvolvimento de uma habilidade de raciocinar recursivamente. Apenas fazendo muitos exercícios podemos dominar a arte de escrever algoritmos recursivos. Seja lá o que "inteligência" venha a ser, recursão está profundamente enraizada nela. No livro *Gödel, Escher, Bach: an Eternal Golden Braid*, Basic Books, 1999 (com 1ª edição de 1979), Douglas Hofstadetr levanta a seguinte questão: o que une as fugas de J.S. Bach, as gravuras de M.C. Escher e as descobertas do matemático Kurt Gödel? Escute Bach, visite a coleção de desenhos de Escher no site do National Gallery of Art (Washington, D.C.) e leia algum resumo da biografia de Kurt Gödel disponível na Internet. Gödel provou o Teorema da Incompleteza, que transformou o mundo da matemática e dos computadores, a partir do entendimento de sistemas autorreferenciados. Entenda essa questão tentando provar o Paradoxo do Cretense Mentiroso:

> "Epimenides, o filósofo nascido em Creta, declara em praça pública: *todo cretense é mentiroso.*"

O que de há de comum entre aqueles três grandes intelectos é a questão da recursão! Encontramos essa questão também no cinema, como na trilogia *De Volta para o Futuro* (Universal Pictures 1985, 1989, 1990). A recursão intriga a humanidade desde a Grécia antiga.

O que fundamenta o processo recursivo é o **princípio da indução matemática** aplicado a problemas com graus crescentes de complexidade: P_m, P_{m+1}, P_{m+2}, …, P_n . Se P_m é verdade, então P_{m+1} também é verdade, desde que saibamos construir a solução para o incremento de 1 grau de complexidade a partir de P_m. Denominamos essa ação incremental com P_m de **passo indutivo**. No caso do fatorial, sabendo que $fat(0) = 1$, encontramos $fat(1) = 1 \times fat(0)$, $fat(2) = 2 \times fat(1)$, …, e assim por diante. A seguinte sistemática (ilustrada para o exemplo do fatorial) pode ser usada a fim de desenvolver uma solução recursiva para um problema *P* qualquer:

1. Observe o domínio do problema e identifique os seus elementos mais neutros (pode ser 0, 1, o string vazio "", uma estrutura vazia, …). Pode haver mais de um elemento neutro a considerar. *Para a função fat() é o* ***0 (zero).***
2. Identifique a versão mais simples do problema *P* cuja solução é trivial (e que, geralmente, está relacionado ao elemento mais neutro do passo 1). Essa versão do problema é chamado de **caso-base**. Pode haver mais de um caso-base. *Para a função fatorial o caso-base é* ***n=0 → fat(n)=1.***
3. Tenha a segurança e a confiança de admitir que a solução do problema *P* com um grau a menos de complexidade (na direção do caso-base) está disponível (*i.e.* é um valor ou um efeito concreto que está disponível). Essa é a **chamada recursiva** (na realidade, é uma hipótese que aceitamos sem questionar qual foi a "mágica" que permitiu a obtenção do valor ou do efeito corretos). Pode haver mais de uma chamada recursiva. *Para a função fatorial é o número inteiro* ***fat(n-1)*** *que admitimos existir e estar disponível.*
4. A solução final vem da declaração de um ou mais **passos indutivos**, em que um passo indutivo é uma modificação simples dos valores ou dos efeitos oriundos das chamadas recursivas (se não for simples é indicação de que estamos no caminho errado). *Para a função fatorial, basta multiplicar a chamada recursiva por n, isto é,* ***fat(n)=n × fat(n-1).***

No caso de funções matemáticas, os passos dessa sistemática são clara e facilmente identificados. Para métodos quaisquer, tal identificação não é tão fácil, mas sempre orienta muito na busca de uma solução recursiva.

O exemplo de um método recursivo para o cálculo do fatorial é apenas didático, pois a sua implementação como processo iterativo (com laços) é mais eficiente e direto. Mas antes de passarmos para problemas cuja solução recursiva é mais adequada, temos de treinar mais situações simples de recursão. O cálculo da série harmônica alternante é um desses casos em que a solução recursiva não representa qualquer ganho, mas serve como treinamento:

$$sAlt(\mathrm{n}) = \sum_{k=1}^{n} \frac{(-1)^{k+1}}{k} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots + \frac{(-1)^{n+1}}{n}$$

Essa série converge para $ln(2) = 0.6931471805599453\ldots$ e a sua implementação, tanto na versão iterativa como na recursiva, está apresentada na Figura 4.7. Procure identificar os quatro passos da sistemática anterior.

```
public static double sAlt(int n)
{
   if (n==1)
      return 1;
   else
      return (Math.pow(-1,n+1)/n)+sAlt(n-1);
}
```

(a) solução recursiva

```
public static double sAlt(int n)
{
   double s=0.0;
   for (;n>=1;n--)
      s += Math.pow(-1,n+1)/n;
   return s;
}
```

(b) solução iterativa

Figura 4.7 Métodos recursivo e iterativo para a série harmônica alternante ***sAlt(n)* = 1 - 1/2 + 1/3 - 1/4 + ...**

O algoritmo de Euclides que calcula o máximo divisor comum (*mdc*) é mais facilmente implementado como um método recursivo do que como um método iterativo (Figura 4.8). No método recursivo basta seguir literalmente este princípio: dados dois números inteiros x e y, se $x > y$, então $mdc(x, y) = mdc(y, r)$ onde r é o resto de $x \div y$.

```
public static int mdc(int x, int y)
{
   if (y==0)
      return x;
   else
      return mdc(y,x%y);
}
```

(a) solução recursiva

```
public static int mdc(int x, int y)
{
   int r=x%y;
   while (r!=0)
   {
      x=y;
      y=r;
      r=x%y;
   }
   return y;
}
```

(b) solução iterativa

Figura 4.8 Métodos recursivo e iterativo para o cálculo do máximo divisor comum (*mdc*) pelo algoritmo de Euclides.

Leonardo de Pisa (ou Leonardo Pisano, em italiano) foi educado em matemática comercial por instrutores hindus, viajou pela costa do Mediterrâneo e aprendeu a maneira como os comerciantes árabes lidavam com aritmética. Leonardo de Pisa passou a ser conhecido por Fibonacci (provavelmente uma corruptela de "filius Bonacci", em que "filius" é "filho" em latim e Bonacci é o plural em italiano do sobrenome Bonaccio de seu pai – um alto funcionário da República de Pisa para assuntos de comércio no norte da África). Fibonacci viveu numa época em que a Europa praticava aritmética com o sistema de números romanos (cuja inexistência do zero tornava qualquer operação um suplício). Fibonacci foi um dos primeiros europeus a introduzir o sistema hindu-arábico na Europa, através de um livro didático que ele propôs para substituir o ensino de aritmética da época (1202). Nesse livro havia um problema matemático (claramente inspirado em similar hindu) cuja solução é hoje conhecida por Série de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, … onde cada termo é a soma dos dois termos anteriores, isto é:

$$a_n = a_{n-1} + a_{n-2}, \text{ para } n \geq 2$$

A série de Fibonacci pode ser encontrada em inúmeras situações na natureza e em computação gráfica, tais como conchas marinhas e texturas dos conjuntos de Mandelbrot. A solução recursiva para a série de Fibonacci parte da identificação do caso-base (*fib*(n) = n para $n = 0$ ou $n = 1$) e da simples soma de duas chamadas recursivas. Por exemplo:

```
public static long fib(long n)
{
   if (n==0 || n==1)
      return n;
   else
      return fib(n-1)+fib(n-2);
}
```

Esse método recursivo gera um processo chamado de **recursão de árvore**, em que cada chamada gera duas outras. Esse processo é muito pouco eficiente, pois o tempo de execução cresce exponencialmente com `n` e as chamadas recursivas são repetidas várias vezes. Certamente a solução iterativa seria bem mais eficiente. Novamente, estamos enfatizando a simplicidade das soluções recursivas. Do ponto de vista de tempo de processamento e uso de memória, os métodos iterativos tendem sempre a ser mais eficientes do que os recursivos. As simples chamadas recursivas dos métodos e a cadeia de operações pendentes são sobrecargas que não existem nos processos iterativos.

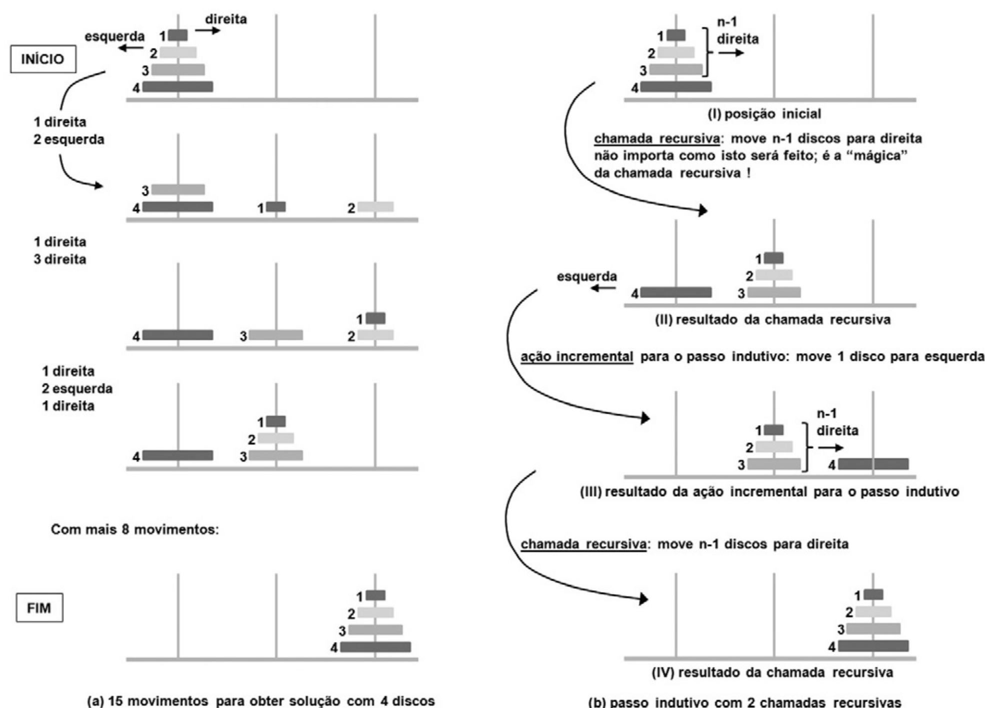


Figura 4.9 (a) ilustração de alguns movimentos para finalizar o jogo; (b) ilustração da definição do processo recursivo.

Uma outra observação importante quando estivermos desenvolvendo soluções recursivas refere-se à tentação de querer simular a cadeia de operações pendentes. Isso não é prático e deve sempre ser evitado. Quando temos segurança em admitir que as chamadas recursivas cumprem suas tarefas corretamente, não necessitamos simular a expansão e a contração da cadeia de operações.

Vamos agora tratar de um problema cuja solução recursiva é muito superior, não pela questão de eficiência, mas pela extrema simplicidade: as **torres de Hanoi**. No quesito eficiência, o processo recursivo da solução desse problema é do tipo recursão de árvore e, portanto, sofre dos males do tempo exponencial. No problema das torres de Hanoi, temos três pinos e n discos de tamanhos diferentes dispostos em determinada ordem no pino mais à esquerda. O desafio é mover todos os n discos para o pino mais à direita, sem mover mais de um disco por vez e sem romper com a ordem relativa entre os discos. Reza a lenda que o mundo acabará quando os monges de um determinado templo hindu acabar de mover 64 discos de ouro para o terceiro pino (uma

tarefa que corresponde a 2^{64}-1 movimentos e que, na velocidade de 1 segundo por movimento, levaria mais de 1.4 bilhões de anos). A Figura 4.9a ilustra alguns passos da solução para quatro discos. A fim de resolver esse problema, vamos identificar os discos 1 a n de cima para baixo. Também vamos considerar que os discos se movem para a esquerda ou para a direita (mover um disco do primeiro pino para a esquerda significa dar a volta, e preencher o terceiro pino e mover um disco para a direita do terceiro pino é voltar para o primeiro pino). Vamos trabalhar com esquerda e direita através de uma variável Booleana que, ao ser negada, significa um movimento ao contrário. Essa variável denominada `esq` começa com verdadeiro (`true`) significando que todos os discos estão à esquerda.

O caso-base é bastante trivial: n = 0 → *não há qualquer movimento*. A estratégia recursiva é a seguinte (Figura 4.9b): a primeira chamada recursiva é supor que os n-1 discos do topo são movidos para a direita (*i.e.*!`esq`), depois operamos no disco do fundo levando-o para a esquerda (é a ação incremental que compõe o passo indutivo) e finalmente admitimos que a chamada recursiva leva os n-1 discos para mais um movimento à direita (*i.e.*!`esq`). A implementação está no Código 4.7. A simples observação desse código revela que o número total de movimentos é dado pela fórmula $T(n) = 2T(n{-}1){+}1$, $T(1) = 1$. A resolução dessa equação é $T(n) = 2^n - 1$, o que pode facilmente ser provado por indução matemática (da mesma maneira que definimos os métodos recursivos). Também podemos provar que esse valor é o número mínimo de movimentos (o que está fora do escopo deste livro). De qualquer maneira, a equação revela o problema do tempo exponencial do algoritmo. Essa preocupação de estimar o tempo de processamento dos algoritmos deve ser constante na vida de um bom programador.

Código 4.7 Jogo Torres de Hanoi

```
import java.util.Scanner;
public class Cap04Ap07      // Torres de Hanoi
{
   public static void move(int n, boolean esq)
   {
      if (n==0)
         return;           // caso-base
      move(n-1,!esq);
      if (esq)
         System.out.println(“move disco “+n+” para ESQUERDA”);
      else
         System.out.println(“move disco “+n+” para direita”);
      move(n-1,!esq);
   }
```

Código 4.7 Jogo Torres de Hanoi (cont.)

```

      public static void main(String[] args)
      {
         Scanner entrada = new Scanner(System.in);
         System.out.print("Entre n: ");
         int n = entrada.nextInt();
         move(n,true);
      }
   }
```

O problema das torres de Hanoi não é apenas um jogo. Uma série de algoritmos com aplicações nas mais variadas áreas são similares ao caso das torres de Hanoi. Jogos, em geral, são ótimos laboratórios para desenvolvimento de novas técnicas.

Nos capítulos seguintes deste livro encontramos outros problemas cuja solução recursiva é mais adequada, simples e direta do que a solução iterativa. Lógicas de jogo mais complexas sempre usam processos recursivos.



Nota
Em algumas linguagens de programação, faz toda a diferença se a chamada recursiva é a última operação a ser realizada. Por exemplo, a chamada recursiva para o cálculo do *mdc* (Figura 4.8a) é imediatamente executada, porque não há operações que a antecedem. Já o mesmo não ocorre com os cálculos do fatorial e dos números de Fibonacci. Chamamos esse tipo de recursão de **recursão de cauda** (***tail recursion***). A princípio, recursão de cauda não precisa gerar uma pilha, porque não são criadas cadeias de operações pendentes. As linguagens dentro do paradigma de programação funcional (*e.g.* Scheme) ou de programação lógica (*e.g.* Prolog) contam com recursão de cauda para gerar processos iterativos. Essas linguagens não precisam de construtores de laço. Na linguagem Java, chamadas recursivas sempre geram pilhas. É típico das linguagens procedimentais, como Java, dependermos dos construtores de laço para implementar processos iterativos. Portanto, não devemos nos preocupar com recursão de cauda no Java.

EXERCÍCIOS

E4.1 Sem usar `if`, escreva o método `boolean intervalo(int x, int a, int b)` que testa se $x \in [a, b]$, onde x, a e b são números inteiros.

E4.2 Sobrecarregue o método do Exercício E4.1 para x, a e b sendo números reais e teste para `intervalo(3,1,5)`, `intervalo(2.5,1,5)` e `intervalo(2,1.5,5)`.

E4.3 O número decimal 0.1 não é representado com precisão pelo sistema binário dos computadores, de maneira que `a==b` é falso se `double a = 1.2;` e `double b = 12*0.1;`. Escreva o método `boolean igual(double x, double y)` que con-

firma x ser igual a y somente se $|x - y| < \text{TOL}$, onde | | significa o módulo e TOL é uma constante pública e estática definida por você. Teste o método para as variáveis `a` e `b` definidas anteriormente.

E4.4 Usando o método auxiliar `long fat(int n)`, escreva um método iterativo que calcula a função: $f(n) = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!}$. Para n suficientemente grande, $f(n)$ se aproxima do valor $e = 2.718281828459045\ldots$, que é a base dos logaritmos naturais.

E4.5 Faça uma versão melhorada do Jogo da Adivinhação (Código 4.6), em que o computador escolhe aleatoriamente os extremos do intervalo $[a, b]$ e diz para o usuário “Pensei em um número entre a e b e você tem dez tentativas para descobrir que número é este”. Limite o tamanho do intervalo a $b - a = 20$ para não prejudicar a jogabilidade.

Capítulo

5

Dados Compostos Como Vetores

Dependendo do problema que queremos resolver utilizando um (ou mais de um) programa(s) de computador, podemos encontrar a situação em que uma grande quantidade de dados precisa ser manipulada simultaneamente em um objeto.

Nesses casos, pode ocorrer de precisarmos relacionar esses dados entre si, e como consequência podemos precisar organizar os dados em um objeto de tal forma que eles possam ser identificados e utilizados com simplicidade e eficiência.

Para dar uma forma mais concreta à situação que estamos tentando caracterizar, consideremos, por exemplo, o que ocorre quando, no início do ano, precisamos organizar nossos recibos e documentos financeiros a fim de montar nossa declaração de renda e propriedades para a Receita Federal.

Nessa situação podemos, por exemplo, ter os recibos escolares indicando o quanto gastamos mensalmente durante todo o ano. Ou seja, teremos doze valores de despesas ordenados e identificados segundo o mês em que cada valor ocorreu.

Se quisermos comparar a despesa de julho com a despesa de agosto, precisaremos ter uma maneira de *apontar* para o valor de julho e recuperar esse valor, e da mesma forma apontar para o valor de agosto e recuperar o valor correspondente àquele mês.

Nesse caso temos um conceito genérico – o conceito de despesa mensal – que tem, em vez de um único valor, doze valores que caracterizam esse conceito

para o ano todo. Cada um dos valores é marcado com um índice – no caso, um valor inteiro que indica de qual mês queremos saber a despesa – e deve poder ser recuperado utilizando o nome do conceito e o índice apropriado. Para comparar as despesas ocorridas em julho e em agosto, por exemplo, devemos ser capazes de recuperar o sétimo e o oitavo valor registrados no conceito genérico *despesa*.

A maioria das linguagens de programação moderna contém recursos sintáticos para representar conceitos genéricos que admitem diversos valores indexados. O nome geral, em inglês, utilizado para essa forma de representar informações é *array*. Em português, o termo mais comumente utilizado para a mesma forma é o nome *vetor*. Assim, vetores em linguagens de programação são formas de representação de conceitos genéricos que admitem diversos valores indexados.

Vejamos como os vetores são representados em Java. Consideraremos, inicialmente, vetores unidimensionais e vetores bidimensionais. A linguagem Java admite, na verdade, vetores com dimensões ainda maiores – tridimensionais, tetradimensionais etc. – mas eles são utilizados mais raramente. De qualquer maneira, tendo entendido com clareza como podemos utilizar os vetores uni e bidimensionais, é relativamente simples extrapolar esse entendimento para vetores com dimensões maiores.

5.1. Vetores unidimensionais

Um vetor unidimensional, em Java, caracteriza um conceito que admite diversos valores, com a restrição de que todos os valores em um vetor devem ter a mesma natureza.

Essa explicação deve se tornar mais clara se considerarmos um exemplo concreto, como o apresentado no Código 5.1. Nesse exemplo, temos a construção de uma classe – a classe `DespesasAnuais` – que contém um vetor com 12 posições, preparadas para conter as despesas correspondentes aos 12 meses de um ano. Esse vetor, denominado `despEduc`, representa o conceito genérico correspondente às despesas com educação. Ele admite 12 valores, sendo que cada um desses valores é do tipo **double**. Em Java, por definição, os valores de um vetor são sempre indexados a partir do zero, portanto, temos que a despesa referente a janeiro será indicada pelo índice 0, a despesa referente a fevereiro será indicada pelo índice 1, e assim por diante.

A construção sintática `double[] despEduc` indica que o conceito `despEduc` será um vetor, cujos valores serão do tipo **double**. Essa construção sin-

tática é completada pela expressão `new double[12]`, que indica que o vetor terá 12 valores e confirma que cada valor será do tipo **double**.

Código 5.1 A Classe DespesasAnuais contém o vetor despEduc

```
class DespesasAnuais
{
   double[] despEduc = new double[12];
   // restante da definição da classe
}
```

O restante da definição da classe pode conter, por exemplo, métodos para manipular o vetor `despEduc`. Por exemplo, podemos ter um método que analisa todos os valores de despesas do ano e retorna o valor máximo, ou seja, a maior despesa com educação ocorrida em um único mês durante o ano considerado. Para fazer isso, é preciso percorrer todos os valores indexados, selecionando dentre eles o maior de todos. Esse método está apresentado no Código 5.2.

Código 5.2 A Classe DespesasAnuais estendida para conter o método maxDesp

```
class DespesasAnuais
{
   double[] despEduc = new double[12];

   double maxDesp()
   {
      int i = 1;
      double max = despEduc[0];
      while (i < 12)
      {
         if (despEduc[i] > max)
         {
            max = despEduc[i];
         }
         i++;
      }
   return max;
   }
   // restante da definição da classe
}
```

Esse método assume por hipótese que a maior despesa ocorreu em janeiro (ou seja, em `despEduc[0]`), e compara esse valor – que fica armazenado na variável `max` – com cada um dos valores indexados e relacionados com o conceito genérico `despEduc` – ou seja, os valores `despEduc[1]`, ..., `despEduc[11]`. Após todas as comparações, o valor máximo encontrado estará armazenado na variável `max`.

Essa classe pode, agora, ser estendida de modo a conter um método **main** para identificar e mostrar ao usuário o valor mensal correspondente ao mês em

que ocorreu a maior despesa com educação. Isso está apresentado no Código 5.3. Observe que, inicialmente, é necessário definir os valores de despesas correspondentes a cada mês, já que não construímos nenhum método para o usuário fornecer esses valores interativamente. Isso pode ser feito utilizando estruturas e padrões vistos em capítulos anteriores, mas neste capítulo fixaremos esses valores dentro do programa, para agilizar e simplificar a produção dos programas. Nesse capítulo, estamos aproveitando a oportunidade para amostrar uma forma simples de definir classes e métodos. No Código 5.3, a classe `DespesasAnuais` tem o vetor `despEdu` disponível em qualquer lugar dentro da classe. Nesse código também criamos uma nova instância desta classe: `DespesasAnuais desp2008 = new Despesas Anuais()`. O operador ponto (.) acessa as propriedades desta instância, que, no caso, são o vetor `despEdu` e o método `maxDesp`. No Capítulo 7 veremos formas mais gerais de definir classes e instâncias (objetos). Uma outra forma de resolver o mesmo problema é definir um método estático que recebe um vetor como argumento: `public static double maxDesp(double[] desp)`. Veja o exercício E5.5.

Um exemplo ligeiramente mais sofisticado permite identificar o *mês* correspondente à maior despesa, em vez de dizer qual foi o valor desta. Para isso, podemos analisar o vetor `despEduc` de maneira similar ao apresentado no Código 5.3, com a diferença que, dessa vez, o método deve retornar como resposta o índice correspondente ao mês de maior valor, em vez do valor propriamente dito. A diferença conceitual importante, nesse caso, é que a resposta do problema não está contida na variável genérica, mas sim em um elemento de sua estrutura – nesse caso, no índice utilizado para organizar seus valores. Esse exemplo está apresentado no Código 5.4, em que foi acrescentado um método para identificar o mês no qual ocorreu o valor máximo, e o método **main** foi alterado para ativar mais esse método.

Código 5.3 A Classe DespesasAnuais estendida com um método main

```
class DespesasAnuais
{
   double[] despEduc = new double[12];

   double maxDesp()
   {
      int i = 1;
      double max = despEduc[0];
      while (i < 12)
      {
         if (despEduc[i] > max)
         {
            max = despEduc[i];
         }
```

Código 5.3 A Classe DespesasAnuais estendida com um método main (cont.)

```
         i++;
      }
   return max;
   }
   public static void main(String[] arg)
   {
      DespesasAnuais desp2008 = new DespesasAnuais();
      desp2008.despEduc[0] = 20.01;
      desp2008.despEduc[1] = 10.11;
      desp2008.despEduc[2] = 22.01;
      desp2008.despEduc[3] = 13.11;
      desp2008.despEduc[4] = 25.01;
      desp2008.despEduc[5] = 19.11;
      desp2008.despEduc[6] = 24.01;
      desp2008.despEduc[7] = 18.11;
      desp2008.despEduc[8] = 23.01;
      desp2008.despEduc[9] = 17.11;
      desp2008.despEduc[10] = 21.01;
      desp2008.despEduc[11] = 16.11;

      double despMax = desp2008.maxDesp();
      System.out.println(despMax);
   }
   // restante da definição da classe
}
```

Código 5.4 A Classe DespesasAnuais estendida para conter o método maxMes

```
class DespesasAnuais
{
   double[] despEduc = new double[12];

   double maxDesp()
   {
      int i = 1;
      double max = despEduc[0];
      while (i < 12)
      {
         if (despEduc[i] > max)
         {
            max = despEduc[i];
         }
         i++;
      }
      return max;
   }
   int maxMes()
   {
      int i = 1;
      double max = despEduc[0];
      int iMax = 0;
      while (i < 12)
      {
         if (despEduc[i] > max)
         {
            max = despEduc[i];
            iMax = i;
         }
         i++;
      }
```

Código 5.4 A Classe DespesasAnuais estendida para conter o método maxMes (cont.)

```
      return iMax;
   }
   public static void main(String[] arg)
   {
      DespesasAnuais desp2008 = new DespesasAnuais();
      desp2008.despEduc[0] = 20.01;
      desp2008.despEduc[1] = 10.11;
      desp2008.despEduc[2] = 22.01;
      desp2008.despEduc[3] = 13.11;
      desp2008.despEduc[4] = 25.01;
      desp2008.despEduc[5] = 19.11;
      desp2008.despEduc[6] = 24.01;
      desp2008.despEduc[7] = 18.11;
      desp2008.despEduc[8] = 23.01;
      desp2008.despEduc[9] = 17.11;
      desp2008.despEduc[10] = 21.01;
      desp2008.despEduc[11] = 16.11;

      double despMax = desp2008.maxDesp();
      System.out.println(despMax);
      int mesMax = desp2008.maxMes();
      System.out.println(mesMax);
   }
   // restante da definição da classe
}
```

Os vetores unidimensionais, portanto, são estruturas organizadas de informação que permitem indexar, armazenar e recuperar valores homogêneos, ou seja, um vetor pode ser utilizado desde que todos os valores armazenados tenham o mesmo tipo – como ocorreu em nosso exemplo ilustrativo, no qual todos os valores armazenados foram do tipo **double**.

Uma possibilidade que enriquece muito a gama de aplicações dos vetores é que os valores homogêneos que "recheiam" um vetor podem ser outros vetores. Ou seja, podemos construir um vetor em que cada um dos valores armazenados seja um novo vetor – um vetor de vetores. Isso permite que valores individuais sejam encontrados utilizando dois índices, de maneira muito parecida com o que ocorre quando, por exemplo, solicitamos a ajuda de um lojista para nos mostrar um produto indicando a posição e a prateleira em que se encontra – "seu Zé, por gentileza me mostre aquele treco que está na *terceira* posição da *segunda* prateleira...", ou seja, o valor que se encontra na terceira posição de um vetor, sendo que esse vetor inteiro é, por sua vez, o valor que se encontra na segunda posição de um "vetorzão" composto por vetores.

Essa possibilidade permite, por exemplo, simular o conceito matemático de matrizes em programas de computador. Quando temos um "vetorzão" composto por vetores, denominamos o "vetorzão" de *vetor bidimensional*. Da

mesma forma, se tivéssemos um "vetorzãozão" composto por "vetorzões", ele seria um *vetor tridimensional*, e assim por diante.

Na próxima seção, detalharemos um pouco mais o conceito de vetores bidimensionais.

5.2. Vetores bidimensionais

Consideremos o problema prático enfrentado por uma empresa de logística, que precisa definir as rotas de seus caminhões de entregas de mercadorias. Essa empresa deve levar em conta diversas questões, como por exemplo, a distância entre cidades, quais caminhos são mais seguros etc.

Consideremos o problema de calcular despesas com pedágio entre cidades. Isso pode ser organizado em uma tabela, de forma que se a empresa a atender *N* cidades a tabela terá *N* linhas e *N* colunas. Os engenheiros de transportes costumam chamar essa tabela de *matriz de origem-destino*, ou *matriz OD*. A matriz é lida da seguinte forma: na linha *i* e na coluna *j* da matriz encontramos o custo referente a partir da cidade *i* e chegar à cidade *j*, assumindo que as cidades foram numeradas e podem ser identificadas por seus índices.

Obviamente, em uma matriz OD, a diagonal principal tem sempre seus valores zerados. Você sabe explicar por quê?

A representação de uma matriz OD em Java pode ser construída utilizando um "vetorzão" de vetores. Cada vetor representará uma coluna da matriz, e o "vetorzão" será, na verdade, um vetor de colunas.

Com isso, podemos, por exemplo, construir um programa que, dada uma matriz OD, informe o custo de pedágio para viajar da cidade de número *i* para a cidade de número *j*. Esse programa está apresentado no Código 5.5.

Inicialmente, incluímos nesse programa a biblioteca `java.util.*`, que contém métodos para a entrada interativa de dados. Com essa biblioteca, podemos fornecer os valores de linha e coluna cujo conteúdo queremos exibir. Em seguida, construímos o "vetorzão" de valores que representarão a matriz OD. Esse "vetorzão" é construído com dois índices. O índice mais à direita indica qual a coluna que queremos analisar, e o índice mais à esquerda indica, dentro de uma coluna, qual o valor específico que queremos recuperar.

Para que a matriz OD tenha valores internos definidos, colocamos arbitrariamente valores de forma que, para posições da matriz diferentes de sua diagonal principal, o custo do pedágio seja igual à soma dos índices da linha

e da coluna. A manipulação de matrizes bidimensionais exige o controle cuidadoso das variáveis de índices, para que seja possível apontar exatamente para cada valor desejado. Vale a pena estudar cuidadosamente, no Código 5.5, como a matriz OD é preenchida com valores numéricos.

Finalmente, temos a parte interativa do programa, na qual o usuário fornece a linha e a coluna de interesse, e o programa apresenta o conteúdo da matriz OD exatamente naquela linha e naquela coluna.

Diversos programas mais sofisticados podem ser construídos a partir desse ponto. Poderíamos, por exemplo, procurar o caminho de menor custo entre duas cidades quaisquer, ou identificar quais cidades podem ser atingidas a partir de uma cidade específica, sem gastar mais do que um valor limite, e assim por diante. Todos esses programas são construídos desenvolvendo métodos que manipulam o “vetorzão” que caracteriza a matriz OD.

Código 5.5 A Classe MatrizOD

```
import java.util.*;

class MatrizOD
{
   double[][] matrizPedagio = new double[10][10];

   public static void main(String[] arg)
   {
      MatrizOD matPed = new MatrizOD();
      int i = 0;
      int j = 0;
      while (i < 10)
      {
         while (j < 10)
         {
            if (i != j)
               matPed.matrizPedagio[i][j] = i + j;
            else
               matPed.matrizPedagio[i][j] = 0.0;
            j++;
         }
         j = 0;
         i++;
      }
      Scanner entrada = new Scanner(System.in);
      System.out.print(“linha: ”);
      int linha = entrada.nextInt();
      System.out.print(“coluna: ”);
      int coluna = entrada.nextInt();
      System.out.print(“pedagio: ”);
      System.out.println(matPed.matrizPedagio[linha][coluna]);
   }
}
```

EXERCÍCIOS

E5.1 Construa um programa que receba dois vetores contendo números inteiros e verifique se o primeiro vetor é subconjunto do segundo vetor.

E5.2 Construa um programa que receba duas matrizes quadradas de tamanho *N* por *N* e calcule a soma dessas duas matrizes.

E5.3 Construa um programa que receba duas matrizes quadradas de tamanho *N* por *N* e calcule o *produto* dessas duas matrizes (dica: você precisará de três índices de controle).

E5.4 Modifique o programa apresentado no Código 5.4, tornando-o interativo, de forma que o conteúdo do vetor despEduc possa ser fornecido pelo usuário.

E5.5 Modifique o Código 5.3 para usar um método estático que recebe o vetor `desp` A estrutura do código é:

```
class DespesasAnuais
{
   public static double maxDesp(double[] desp)
   {
   ...
   }
   public static void main(String[] args)
   {
      double[] despEdu = newdouble[12]
      despEdu[0] = 20.01;
      ...
      double despMax = masDesp(despEdu);
   ...
   }
}
```

Capítulo

6

Usando Mais Objetos e Classes Simples

Neste capítulo vamos enfatizar a construção de programas contendo diversos objetos, que podem interagir e dessa forma produzir efeitos interessantes. Um objeto é uma instância de uma classe.

A base da construção de programas orientada a objetos é a construção de entidades com certo grau de autonomia – ou seja, objetos – construídos a partir do que estiver especificado em suas respectivas classes e capazes de reagir a solicitações externas e oferecer serviços, de forma que, cooperativamente (quando coordenados de forma apropriada) os objetos de um sistema possam resolver problemas, oferecer serviços aos usuários, apresentar efeitos interessantes etc.

Por exemplo, vamos considerar neste capítulo que um biólogo deseje simular interações entre animais em cativeiro. Para isso, o biólogo pode construir uma classe para cada animal observado, posteriormente construir objetos que representem indivíduos específicos de cada classe, e finalmente utilizar um objeto de coordenação que provoque atividades pertinentes para cada animal e registre o resultado dessas atividades e interações entre os animais. No Capítulo 7 veremos formas mais gerais de classes e objetos.

6.1. Um programa, muitas classes...

Nosso biólogo deve observar o comportamento de animais em um zoológico. Mais especificamente, lhe foram destinados os seguintes animais para observação:

- Dois rinocerontes;
- Um antílope;
- Três hipopótamos.

Ou seja, o biólogo tem sob seus cuidados dois indivíduos da classe "rinoceronte", um indivíduo da classe "antílope" e três indivíduos da classe "hipopótamo".

Se nosso biólogo quiser construir um programa para simular o comportamento dos animais, ele pode construir uma classe para cada espécie sob seus cuidados. Cada classe poderá conter características e operações distintas, pois cada uma representa uma espécie diferente.

Vamos construir um exemplo bem simplificado, apenas para ilustrar como isso poderia ser codificado e utilizado em Java. Se não mostrarmos nosso programa para nenhum biólogo profissional, tudo deverá correr bem.

Em nosso exemplo simplificado, os animais fazem somente três coisas na vida: andam, comem e dormem. Para diferenciar um pouco as espécies, vamos considerar que um antílope sempre corre a 20 km/h, um hipopótamo sempre anda a 0,2 km/h e um rinoceronte caminha a uma velocidade que depende de seu peso: considerando que o peso de um rinoceronte seja fornecido em toneladas, a sua velocidade de caminhada é dada pela expressão $vel_{rino} = 1/peso_{rino}$.

Para cada espécie, podemos ter uma classe distinta, conforme apresentado nos Códigos 6.1, 6.2 e 6.3. Deve ser observado que cada classe tem um *construtor*, que é o componente que recebe o mesmo nome da classe e caracteriza o que deve ocorrer no momento em que um objeto daquela classe é produzido. No caso da classe *Rinoceronte*, por exemplo, é preciso fornecer o peso do rinoceronte para construir um animal específico, pois essa informação é necessária a fim de determinar aspectos de seu comportamento (no nosso exemplo simplificado, para determinar a velocidade de deslocamento do rinoceronte).

Uma vez construídas as classes que representam as espécies, podemos construir uma classe de coordenação, para criar cada um dos animais indi-

viduais (dois rinocerontes, um antílope e três hipopótamos) e disparar cada ação de cada animal. Em nosso pequeno exemplo, as ações não interferem umas nas outras e ocorrem quase instantaneamente, mas já são suficientes para demonstrar como os objetos construídos a partir de classes distintas podem conviver e atuar em conjunto em um mesmo programa.

Código 6.1 A Classe Rinoceronte

```
public class Rinoceronte
{
   double Peso;

   Rinoceronte(double p)
   {
      Peso = p;
   }

   void andar()
   {
      System.out.println(“Rino andando: “ + 1/Peso + “km/h”);
   }

   void comer()
   {
      System.out.println(“Rino comendo”);
   }

   void dormir()
   {
      System.out.println(“Rino dormindo”);
   }

}
```

Código 6.2 A Classe Antilope

```
public class Antilope
{
   Antilope()
   {
   }

   void andar()
   {
      System.out.println(“Anti correndo”);
   }

   void comer()
   {
      System.out.println(“Anti comendo”);
   }

   void dormir()
   {
      System.out.println(“Anti dormindo”);
   }

}
```

Código 6.3 A Classe Hipopotamo.

```
public class Hipopotamo
{
   Hipopotamo()
   {
   }

   void andar()
   {
      System.out.println("Hipo andando");
   }

   void comer()
   {
      System.out.println("Hipo comendo");
   }

   void dormir()
   {
      System.out.println("Hipo dormindo");
   }

}
```

A classe de coordenação pode ficar conforme apresentado no Código 6.4.

Código 6.4 A Classe Coordenador

```
class Coordenador
{
   public static void main(String[] arg)
   {
      Rinoceronte rino1 = new Rinoceronte(10);
      Rinoceronte rino2 = new Rinoceronte(12);
      Antilope anti = new Antilope();
      Hipopotamo hipo1 = new Hipopotamo();
      Hipopotamo hipo2 = new Hipopotamo();
      Hipopotamo hipo3 = new Hipopotamo();

      rino1.dormir();
      anti.comer();
      hipo1.dormir();
      hipo1.andar();
      hipo2.andar();
      hipo3.comer();
      rino2.andar();
   }
}
```

Essa classe constrói seis objetos: dois da espécie Rinoceronte, um da espécie Antílope e três da espécie Hipopótamo. Em seguida, ela determina a sequência de ações correspondentes a cada um desses objetos individuais construídos.

Em nosso exemplo simplificado, cada ação é registrada com uma frase no dispositivo de saída. Um defeito do nosso sistema, entretanto, é que ele não permite identificar individualmente os responsáveis pelas ações efetuadas: ao encontrar no dispositivo de saída a sentença "*Hipo dormindo*", por exemplo, não sabemos se quem está dormindo é o Hipopótamo *hipo1*, o *hipo2* ou o *hipo3*.

Para complementar o exemplo, precisamos ser capazes de identificar cada objeto criado com um nome. Para isso, é preciso contar, dentro de cada objeto, com uma variável cujo conteúdo seja composto por cadeias de caracteres, ou seja, uma variável cujo tipo de dados seja String.

6.2. O tipo de dados String

Em capítulos anteriores, vimos os oito tipos de dados primitivos da linguagem Java. Vimos também, brevemente, o tipo de dados String, utilizado para representar cadeias de caracteres – ou seja, textos (em geral breves), como sentenças, ordens, observações, nomes etc. – com a ressalva de que, em Java, esse tipo de dados não é um tipo primitivo.

Mais especificamente, em Java *não temos* propriamente um tipo de dados String. Para manter essa linguagem concisa, os engenheiros que a projetaram e construíram preferiram deixar o recurso de manipulação de dados caracterizados como cadeias de caracteres como um recurso opcional, que pode ser incluído em um programa conforme a necessidade. Para fazer isso, foi construída uma *classe* String, cujos métodos implementam as operações que, em geral, desejamos efetuar com cadeias de caracteres. Em vez de construir uma variável e vinculá-la a um tipo de dados primitivo, para representar e manipular uma cadeia de caracteres, construímos um objeto da classe String.

Em termos práticos, a diferença é que, para efetuar operações com cadeias de caracteres em Java, precisamos acionar métodos presentes na classe String, de quem cada cadeia específica de caracteres será um objeto. Por exemplo, se criarmos uma cadeia de caracteres com o comando `String s = "abcdef"`, teremos um objeto da classe String com o nome `s` em nosso programa. Os caracteres em uma cadeia de caracteres em Java são numerados, iniciando do zero. Assim, o "`a`" na cadeia `s` é o caractere de número zero, o "`b`" é o caractere de número um, e assim por diante. Se criarmos uma segunda cadeia de caracteres com o comando `String ss = s.substring(2)`, estaremos utilizando

o método `substring`, que faz parte da classe String, para construir uma segunda cadeia de caracteres. O conteúdo da cadeia de caracteres `ss` é o trecho da cadeia `s`, iniciando a partir do caractere de número dois e indo até o final, ou seja, "`cdef`".

Alguns métodos frequentemente utilizados, dentre os muitos disponíveis na classe String, são os seguintes:

- `s.substring(x)`: fornece a cadeia de caracteres iniciando no caractere de número `x` de `s` e indo até o final de `s`.
- `s.substring(x, y)`: fornece a cadeia de caracteres iniciando no caractere de número `x` de `s` e indo até o caractere de número `y-1`.
- `s.length()`: fornece um valor inteiro que corresponde à quantidade de caracteres de `s`.
- `s.toUpperCase()`: fornece a cadeia de caracteres similar à cadeia `s`, porém convertendo todas as letras para caixa alta.
- `s.toLowerCase()`: fornece a cadeia de caracteres similar à cadeia `s`, porém convertendo todas as letras para caixa baixa.
- `s.compareTo(t)`: compara as cadeias de caracteres `s` e `t`. Se elas forem idênticas, retorna o valor zero; se `s` vier alfabeticamente antes de `t`, retorna um inteiro menor que zero; se `s` vier alfabeticamente depois de `t`, retorna um inteiro maior que zero.
- `s.indexOf('c')`: fornece o número da primeira ocorrência do caractere "`c`" na cadeia `s`.
- `s.indexOf('c',n)`: fornece o número da primeira ocorrência do caractere "`c`" na cadeia `s`, que seja maior que ou igual a `n`.
- `s = s1 + s2`: coloca na cadeia `s` o resultado da concatenação das cadeias `s1` e `s2`. Por exemplo, se `s1 = "abc"` e `s2 = "def"`, então `s = "abcdef"`.

A classe String pode ser usada para dar nomes aos animais no exemplo do biólogo, dessa forma identificando qual animal está efetuando qual ação. Essa possibilidade está ilustrada no Código 6.5.

Código 6.5 Exemplo de uso da classe String

```
public class Hipopotamo
{
   String nome;
   Hipopotamo(String n)
   {
      nome = n.substring(5);
   }
   void andar()
```

Código 6.5 Exemplo de uso da classe String (cont.)

```
   {
      System.out.println(nome + “ andando”);
   }

   void comer()
   {
      System.out.println(nome + “ comendo”);
   }

   void dormir()
   {
      System.out.println(nome + “ dormindo”);
   }
}

public class Antilope
{
   String nome;
   Antilope(String n)
   {
      nome = n.substring(5);
   }
   void andar()
   {
      System.out.println(nome + “ correndo”);
   }

   void comer()
   {
      System.out.println(nome + “ comendo”);
   }

   void dormir()
   {
      System.out.println(nome + “ dormindo”);
   }
}
public class Rinoceronte
{
   double Peso;
   String nome;
   Rinoceronte(double p, String n)
   {
      Peso = p;
      nome = n.substring(5);
   }
   void andar()
   {
      System.out.println(nome+” vai a “ + 1/Peso + “km/h”);
   }
   void comer()
   {
      System.out.println(nome + “ comendo”);
   }
   void dormir()
   {
      System.out.println(nome + “ dormindo”);
   }
}

```

Código 6.5 Exemplo de uso da classe String (cont.)

```
class Coordenador
{
   public static void main(String[] arg)
   {
      Rinoceronte rino1 = new Rinoceronte(10,”Rino rino1”);
      Rinoceronte rino2 = new Rinoceronte(12, “Rino rino2”);
      Antilope anti = new Antilope(“Anti anti”);
      Hipopotamo hipo1 = new Hipopotamo(“Hipo hipo1”);
      Hipopotamo hipo2 = new Hipopotamo(“Hipo hipo2”);
      Hipopotamo hipo3 = new Hipopotamo(“Hipo hipo3”);

      rino1.dormir();
      anti.comer();
      hipo1.dormir();
      hipo1.andar();
      hipo2.andar();
      hipo3.comer();
      rino2.andar();
   }
}
```

Diversos usos mais elaborados da classe String podem ser imaginados. Esse pequeno exemplo zoológico serve para ilustrar alguns conceitos fundamentais e fornecer os fundamentos para que os usos mais elaborados possam ser construídos.

Se elaborarmos mais esse exemplo, pode ser que precisemos obter dados de entrada de um arquivo de configuração, e/ou registrar o resultado de nossas simulações em um arquivo de saída, para posterior consulta e análise. Para podermos efetuar isso, é preciso aprendermos a manipular arquivos. Na próxima seção veremos recursos simples para leitura e gravação de arquivos.

6.3. Leitura e gravação de arquivos

Em diversas circunstâncias, pode ser útil saber manipular arquivos de dados. Por exemplo, podemos precisar de dados de configuração para inicializar a operação de um programa, ou registrar o resultado de computações em um arquivo de saída.

Os recursos necessários para leitura e gravação de arquivos se encontram em classes disponíveis no pacote `java.io`, que deve ser importado para o programa construído de modo a disponibilizar esses recursos de maneira mais prática. Para isso, devemos incluir a linha “`import java.io.*;`” no arquivo que define a classe em que estarão presentes os comandos de manipulação de arquivos.

A abertura de um arquivo para leitura deve ocorrer em três etapas: inicialmente, deve ser definida a fonte de dados, como um objeto de `FileInputStream`. Em seguida, deve ser definido um repositório de leitura, parametrizado pela fonte de dados, como um objeto de `InputStreamReader`. Finalmente, deve ser definido um leitor desse repositório, como um objeto de `BufferedReader`. Esse último objeto tem um método denominado `readLine()`, que retorna uma cadeia de caracteres, a qual pode ser capturada em uma variável.

A abertura de um arquivo para gravação segue o caminho inverso ao da abertura para leitura: inicialmente, deve ser definido um destino para os dados, como um objeto de `FileOutputStream`. Em seguida, deve ser definido um repositório de gravação, parametrizado pelo destino de dados, como um objeto de `OutputStreamWriter`. Finalmente, deve ser definido um gravador desse repositório, como um objeto de `BufferedWriter`. Esse último objeto tem métodos como `write(s)`, que grava a cadeia de caracteres `s` no arquivo de gravação, e `newLine()`, que grava um caractere de mudança de linha no arquivo de gravação.

No Código 6.6 ilustramos o uso desse procedimento para leitura e gravação de dados em arquivos. Acrescentamos em nossa classe de coordenação a leitura do nome do biólogo de plantão, presente em um arquivo de configuração, e a gravação de duas mensagens em um arquivo de saída, a primeira indicando o início da simulação e a segunda indicando o final. Se, no mesmo diretório de arquivos em que se encontra o programa em execução, estiverem presentes os arquivos `entrada.txt` e `saída.txt`, sendo que na primeira linha do arquivo `entrada.txt` estiver presente o nome do biólogo, após a execução do programa devem estar gravadas duas linhas no arquivo de saída, uma dando boas-vindas ao biólogo e a segunda indicando o final da simulação.

Código 6.6 A Classe Coordenador com leitura e gravação de arquivos

```
import java.io.*;
class Coordenador
{
   public static void main(String[] arg) throws IOException
   {
      InputStream is = new FileInputStream("entrada.txt");
      InputStreamReader isr = new InputStreamReader(is);
      BufferedReader br = new BufferedReader(isr);

      OutputStream os = new FileOutputStream("saida.txt");
      OutputStreamWriter osw = new OutputStreamWriter(os);
      BufferedWriter bw = new BufferedWriter(osw);
```

Código 6.6 A Classe Coordenador com leitura e gravação de arquivos (cont.)

```
            String linha = br.readLine();
            bw.write(“Ola ” + linha);

            Rinoceronte rino1 = new Rinoceronte(10,”Rino rino1”);
            Rinoceronte rino2 = new Rinoceronte(12, “Rino rino2”);
            Antilope anti = new Antilope(“Anti anti”);
            Hipopotamo hipo1 = new Hipopotamo(“Hipo hipo1”);
            Hipopotamo hipo2 = new Hipopotamo(“Hipo hipo2”);
            Hipopotamo hipo3 = new Hipopotamo(“Hipo hipo3”);

            rino1.dormir();
            anti.comer();
            hipo1.dormir();
            hipo1.andar();
            hipo2.andar();
            hipo3.comer();
            rino2.andar();

            bw.newLine();
            bw.write(“Ate mais tarde ” + linha);
            bw.close();
        }
    }
```

6.4. Classes abstratas e interfaces

Em nosso exemplo zoológico, temos as três classes `Rinoceronte`, `Antilope` e `Hipopotamo` utilizadas no mesmo programa para a simulação do comportamento de animais. Claramente, as três classes são casos particulares de uma classe mais genérica, que poderíamos denominar `Mamífero`, por exemplo.

A construção de uma hierarquia de conceitos permite organizar melhor os programas. Podemos, por exemplo, construir a definição parcial de uma classe mais genérica, e posteriormente estender e especializar essa classe para os casos mais específicos. Dependendo do caso, também é possível apenas especificar o conteúdo esperado de uma classe concreta que implemente um conceito genérico.

Em Java temos suporte para essas duas possibilidades, através dos conceitos de *classe abstrata* e *interface*, respectivamente. Uma classe abstrata é a construção parcial de um conceito genérico, que pode conter objetos e métodos que estarão presentes em todos os casos particulares. A classe abstrata pode, então, ser complementada com objetos e métodos adicionais, ou ter alguns de seus métodos modificados dentro dos casos particulares para melhor atender situações específicas de cada caso. Uma classe mais concreta estende (*extends*) a classe abstrata.

Por outro lado, uma interface, por definição, não contém implementações de métodos ou objetos, somente a forma como esses métodos se co-

municam com quem solicita suas ações. Uma interface especifica o que deve conter qualquer uma de suas implementações, sem, entretanto, determinar como esse conteúdo deve ser implementado. Interfaces não têm construtores nem podem ser instanciadas com o operador new.

Outra diferença importante entre classes abstratas e interfaces é que podemos construir uma classe que implementa simultaneamente diversas interfaces – ou seja, constitui um conceito que caracteriza todas as propriedades e operações previstas em todas as interfaces que ele implementa – mas uma classe pode ser a extensão de uma única classe abstrata – ou seja, nenhuma classe pode simultaneamente estender duas ou mais classes.

Em geral, é relativamente simples e intuitivo identificar se um conceito genérico deve ser caracterizado em um sistema como uma classe abstrata ou como uma interface. Em alguns casos específicos, é possível escolher dentre essas duas possibilidades de forma relativamente arbitrária.

Em nosso exemplo zoológico, podemos generalizar o conceito de `Mamífero` como uma classe abstrata ou como uma interface. No Código 6.7 apresentamos a classe abstrata `Mamífero` e como ficaria a classe `Rinoceronte` se ela fosse apresentada como extensão da classe abstrata `Mamífero`. Nesse código, a palavra-chave `@Override` ajuda o compilador a lidar com a sobrecarga de métodos. No Código 6.8 apresentamos a interface `Mamífero` e como ficaria a classe `Rinoceronte` se ela fosse uma implementação dessa interface. Deixamos como exercício completar cada uma dessas duas possibilidades com as classes `Antilope` e `Hipopotamo`.

Código 6.7 A Classe Abstrata Mamifero e a Classe Rinoceronte como uma extensão

```
public abstract class Mamifero
{
   Mamifero()
   {
   }

   void andar()
   {
      System.out.println("Mamifero andando");
   }

   void comer()
   {
      System.out.println("Mamifero comendo");
   }

   void dormir()
   {
      System.out.println("Mamifero dormindo");
```

Código 6.7 A Classe Abstrata Mamifero e a Classe Rinoceronte como uma extensão (cont.)

```
   }

}

public class Rinoceronte extends Mamifero
{
   double Peso;

   Rinoceronte(double p)
   {
      Peso = p;
   }

   @Override
   void andar()
   {
      System.out.println("Rino andando: " + 1/Peso + "km/h");
   }

   @Override
   void comer()
   {
      System.out.println("Rino comendo");
   }

   @Override
   void dormir()
   {
      System.out.println("Rino dormindo");
   }

}
```

Código 6.8 A Interface Mamifero e a Classe Rinoceronte como sua implementação

```
interface Mamifero
{
   void andar();
   void comer();
   void dormir();
}

public class Rinoceronte implements Mamifero
{
   double Peso;

   Rinoceronte(double p)
   {
      Peso = p;
   }

   public void andar()
   {
      System.out.println("Rino andando: " + 1/Peso + "km/h");
   }

   public void comer()
   {
      System.out.println("Rino comendo");
```

Código 6.8 A Interface Mamifero e a Classe Rinoceronte como sua implementação (cont.)

```
        }

        public void dormir()
        {
            System.out.println("Rino dormindo");
        }

    }
```

EXERCÍCIOS

E6.1 Construa um programa similar ao do biólogo (apresentado nas seções anteriores), para simular a interação de robôs inteligentes em um cenário de ficção científica. Defina as ações para cada um dos robôs e efetue simulações das interações entre esses robôs.

E6.2 Modifique o seu programa do Exercício E6.1, para que agora ele leia de um arquivo de configuração quantos robôs de cada tipo estarão presentes em sua simulação. A simulação deverá levar em consideração a existência de diversos exemplares de cada tipo de robô disponível para o seu programa.

E6.3 Modifique novamente o seu programa, agora para que o resultado das suas simulações fique registrado em um arquivo de saída.

E6.4 Reconstrua seu programa, agora utilizando uma classe abstrata para o conceito de *Mamífero*, que deve ser estendida para cada animal específico.

E6.5 Refaça o exercício anterior, mas agora utilizando uma interface para o conceito de *Mamífero*, que deve ser implementada para cada animal específico.

Capítulo

7

Classes, Objetos e Herança

Programar com orientação a objetos corresponde a modelar o mundo como uma coleção de entidades (chamadas objetos) que têm propriedades e comportamentos próprios e que colaboram entre si. Dessa maneira, o mundo fica mais modularizado e fácil de entender. Objetos podem ser compostos por outros objetos, formando entidades mais complexas. Objetos podem ser vistos como entidades concretas criadas a partir de classes genéricas, num processo que chamamos de instanciação. Neste caso, um objeto concreto é uma instância de alguma classe. Por exemplo, rino1 é uma instância da classe dos rinocerontes, um jogador (*Player*) é uma instância da classe dos objetos de jogo (*Game Objects*) do motor de jogos deste livro, e você é uma instância única e concreta da classe dos humanos. Cada objeto é responsável pela realização de um conjunto de tarefas a ele inerentes. Se um objeto precisar de uma tarefa que não é de sua responsabilidade, ele pede a um outro objeto, através de uma mensagem, que realize essa tarefa.

No Capítulo 6 já apresentamos o conceito inicial da construção de programas usando objetos. Neste capítulo, vamos aprender a criar classes e objetos com mais rigor e detalhes. Também vamos lidar com um conceito fundamental para a programação orientada a objetos: herança.

7.1. Classes e objetos

Uma **classe** é um molde (*template*) que define, de maneira genérica, como serão os objetos. Uma classe é como uma receita para construir objetos. Objetos são criados como sendo **instâncias** de uma classe. Uma classe tem a anatomia mostrada na Figura 7.1, que inclui nome, propriedades (definidas por variáveis) e comportamentos (realizados através de métodos) que caracterizarão os objetos. As propriedades, também denominadas **campos de dados** (***data fields***), podem ser entendidas como sendo atributos que caracterizam a estrutura de um objeto. Os comportamentos são ações e tarefas inerentes aos objetos criados a partir da classe e que usam as propriedades definidas. Dessa maneira, um objeto leva com ele a informação e o conhecimento necessários para o seu desempenho numa "sociedade" de objetos.

Nome
Propriedades (variáveis)
Comportamentos (métodos)

Figura 7.1 Anatomia geral de uma classe.

Entre os comportamentos de uma classe há métodos especiais, denominados **construtores** (***constructors***), que são chamados toda vez que um novo objeto é criado. Os construtores têm o papel de inicializar os objetos. Os outros métodos podem ser chamados de **métodos de instância** ou simplesmente métodos. A Figura 7.2 ilustra uma classe `Retangulo`, com posição e dimensões medidas em pixels, que tem dois construtores e dois métodos (um que retorna a área e outro que translada a origem em incrementos). Ao criarmos um novo objeto `Retangulo`, devemos escolher um dos construtores (o primeiro que usa sempre os mesmos valores para as variáveis ou o segundo que nos obriga a fornecer esses valores). As restrições com relação a construtores são as seguintes: o nome deve ser o mesmo da classe (`Retangulo`, no caso) e não têm tipo de retorno (nem mesmo `void`). O método `getArea` calcula e retorna a área do retângulo, e o método `transladar` move o retângulo de um incremento dado. Os atributos de um objeto `Retangulo`, por serem públicos (`public`), podem ser alterados diretamente, como veremos a seguir.

O Código 7.1 apresenta um programa que testa a classe `Retangulo`. Construtores são invocados usando o operador `new`. As variáveis que representam os objetos criados são denominados **variáveis de referência** (esse tipo de variável contém uma referência ao objeto, em vez de conter o objeto em si). O resultado desse teste (ilustrado na Figura 7.3) é o seguinte:

```
Area r1= 50
origem r2: (10,15)
```

onde `r1` e `r2` são variáveis de referência.

Depois que um objeto é criado, usamos o **operador ponto** (***dot operator***) para acessar/alterar os seus dados (*i.e.* propriedades), como em `r1.x` na linha 9 do Código 7.1, ou para invocar os seus métodos, como em `r2.transladar(5, 0)` na linha 12.

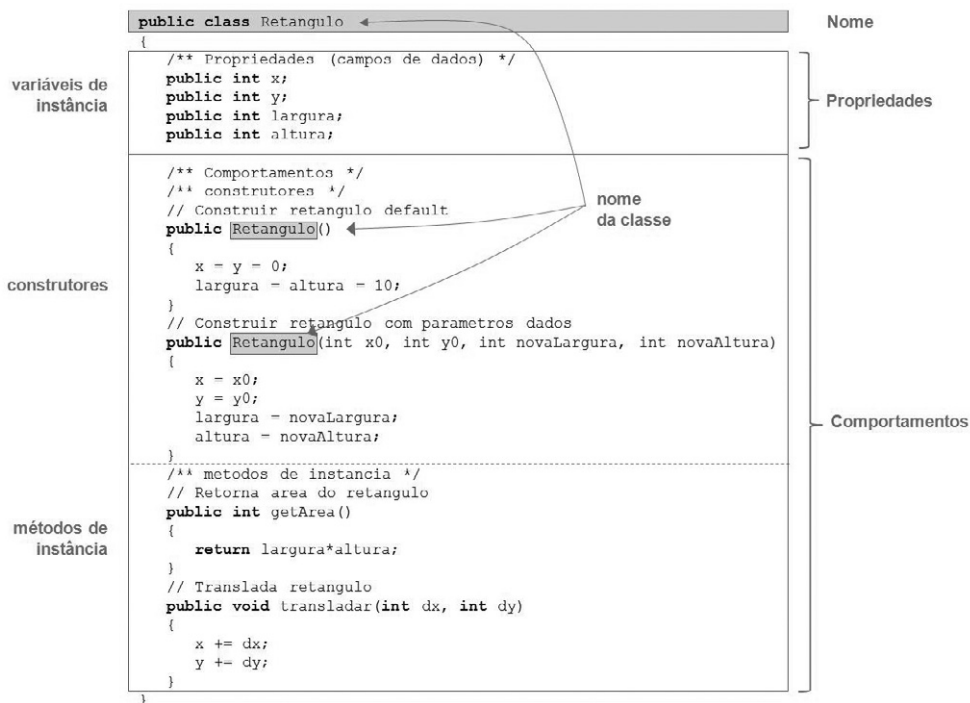


Figura 7.2 Exemplo de uma classe com seus elementos principais: nome, propriedades e comportamentos. Construtores devem ter o mesmo nome da classe e não tem tipo de retorno (nem mesmo `void` é permitido).

Há várias maneiras de escrever um programa Java completo que testa uma classe. Neste livro, vamos trabalhar com arquivos separados, um para cada classe. A Figura 7.4 apresenta um projeto chamado `TesteRetangulo`, no ambiente NetBeans, que contém duas classes: `Main.java` (Código 7.1) e `Retangulo.java` (Figura 7.2). Neste livro, sempre trabalhamos com pacotes (*packages*). Por essa razão, a primeira linha do Código 7.1 é a diretiva `package testeretangulo;` que também deve ser colocada no código completo da classe `Retangulo.java` (Figura 7.2). Toda classe em Java pertence a um pacote. Quando não colocamos uma diretiva `package`, a classe passa a fazer parte do pacote default sem nome (*unnamed package*) – o que não recomendamos.

Código 7.1 Teste da classe Retangulo (Figura 7.2)

```
package testeretangulo;

public class Main
{
   public static void main(String[] args)
   {
      Retangulo r1 = new Retangulo(); // Cria retangulo r1
      Retangulo r2 = new Retangulo(0,15,5,10); // Cria retangulo r2
      r1.x = 15;       // altera origem de r1
      r1.altura = 5;  // altera altura de r1
      System.out.println("Area r1= " + r1.getArea());
      r2.transladar(5, 0); // translada origem de 5 pixels na horizontal
      r2.transladar(5, 0); // translada origem de mais 5 pixels
      System.out.println("origem r2: (" + r2.x + "," + r2.y + ")");
   }
}
```

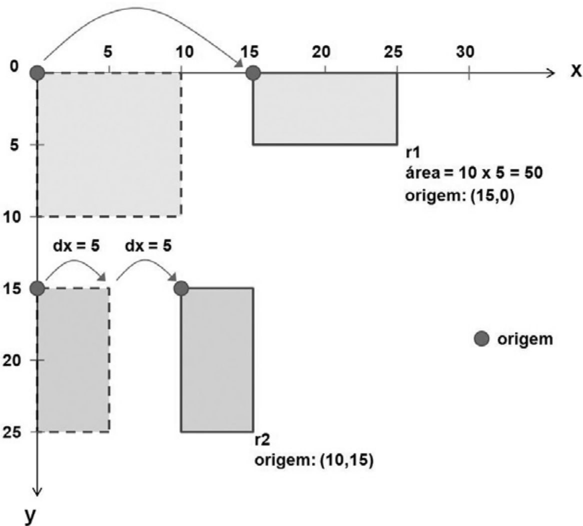


Figura 7.3 Alterações nos retângulos `r1` e `r2` conforme Código 7.1.



Figura 7.4 Projeto `TesteRetangulo` no ambiente NetBeans contendo a classe `Retangulo` (Figura 7.2) e a classe `Main` que a testa (Código 7.1). A diretiva `package testeretangulo`;deve ser adicionada ao código da Figura 7.2.

Projetar classes e objetos requer diagramas que ilustrem o conteúdo de cada classe e a maneira como os objetos se relacionam. A notação mais usada é a estabelecida pelo padrão UML (*Unified Modeling Language*), que denominamos diagramas de classe UML (ver www.uml.org). Nesses diagramas, os campos de dados são indicados por *nome*: *tipo* e métodos por *nome* (*parâmetro*: *tipo*): *tipoDeRetorno*. Além disso, o prefixo + indica variáveis ou métodos públicos (*public*), o prefixo – denota um modificador privado (*private*) e sublinhado indica variáveis ou métodos estáticos (*static*). A Figura 7.5 apresenta o diagrama de classe para `Retangulo`.

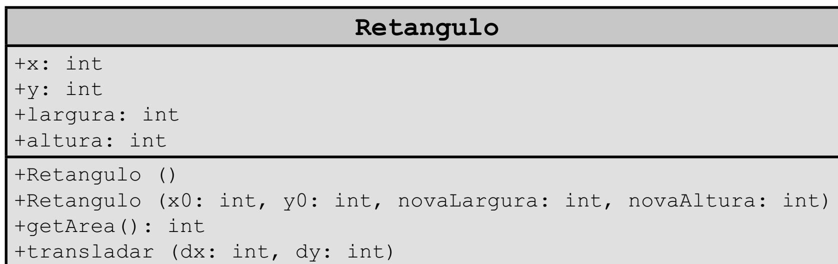


Figura 7.5 Diagrama de classe UML da classe `Retangulo` (Figura 7.2), onde + denota `public`. Neste exemplo não há situações `private` nem `static`.

A rigor, os campos de dados e os métodos de uma classe podem ter outros tipos que não são instâncias, assunto tratado na Seção 7.3.

7.2. O modificador *static*

Código 7.2 Classe Particula com variável estática, constante e método estático

```
public class Particula
{
   public static final double G = 9.807;  // aceleracao da gravidade
   public double massa;
   public static int Nparticulas = 0;

   public Particula(double m0)
   {
      massa = m0;
      Nparticulas++;
   }

   public double peso()
   {
      return massa * G;
   }
   public static int getNparticulas()
   {
      return Nparticulas;
   }
}
```

Código 7.3 Teste da classe Particula ilustrando as consequências do modificador static

```
public class Main
{
   public static void main(String[] args)
   {
      System.out.println(“No. de particulas: ‟ + Particula.
         Nparticulas);
      System.out.println(“G= ‟ + Particula.G);
      System.out.println(“No. de particulas:
         ‟ + Particula.getNparticulas());
      Particula p1 = new Particula(10.0);
      Particula p2 = new Particula(2.0);
      System.out.println(“No. de particulas:
         ‟ + p1.Nparticulas);
      System.out.println(“No. de particulas:
         ‟ + Particula.getNparticulas());
   }
}
```

No Código 7.2 temos uma classe `Particula` na qual a variável `Nparticulas` e o método `getNparticulas` são estáticos. Quando usamos o modificador `static` numa variável de instância, se um objeto atualiza o valor dessa variável, então todos os objetos são afetados (isto é: a variável estática é compartilhada por todos os objetos da classe). Tanto a variável estática como o método estático podem ser acessados por uma variável de referência ou pelo nome da classe. Mesmo sem nenhum objeto ter sido criado, o acesso a variáveis e métodos estáticos pelo nome da classe é sempre possível. Uma constante (como a `G` do Código 7.2) também é compartilhada por todos os objetos e pode ser acessada pelo nome da classe. O Código 7.3 ilustra todas essas situações, produzindo o seguinte resultado:

```
No. de particulas: 0
G = 9.807
No. de particulas: 0
No. de particulas: 2
No. de particulas: 2
```

7.3. Instância *versus* estático

Após o entendimento do modificador static (Seção 7.2), fica claro por que usamos o termo instância para a maior parte das variáveis e dos métodos de uma classe. Variáveis de instância e métodos de instância são variáveis e métodos que dependem de instâncias específicas de uma classe. Por exemplo, na definição da classe `Particula` (Código 7.2), consideramos que toda a partí-

cula tem sua própria massa, isto é, massa é dependente de uma partícula específica. Portanto, `massa` deve ser uma variável de instância. Da mesma forma, o método `peso` é dependente de uma partícula específica e deve, portanto, ser um método de instância. Ao projetar uma classe, você deve sempre identificar quais variáveis e métodos não dependem de uma instância específica – estes devem ser definidos como estáticos. Por exemplo, a variável `Nparticulas` e o método `getNparticulas` da classe `Particula` são definidos como estáticos.

Durante a definição de uma classe, não permita que variáveis e métodos de instância sejam usados a partir de métodos estáticos (porque esses últimos não podem depender de instâncias específicas). Por exemplo, você não pode usar a variável de instância `massa` dentro do método estático `getNparticulas()`. Nesse particular, você deve notar que métodos bem conhecidos da classe `Math`, tais como `sin` e `cos`, são estáticos e podem ser chamados livremente (inclusive pelo método estático `main`).

7.4. Abstração de classe e encapsulamento

Um dos conceitos mais importantes na ciência da computação é abstração. Quando usamos um artefato através de sua interface com o mundo externo (isto é, através de sua funcionalidade), sem a necessidade de saber como é o funcionamento interno desse artefato, estamos praticando a abstração. Você está abstraindo quando usa um avião para se transportar sem precisar saber como ele funciona internamente. A situação é similar com os artefatos lógicos da computação. Nesse particular, **abstração de classe** é a separação do uso de uma classe da implementação dessa classe. Nesse caso, o projetista de uma classe provê uma descrição desta e deixa o usuário saber como ela pode ser usada, sem revelar os detalhes de sua implementação. A classe é, então, usada através da coleção de métodos e constantes públicos junto com a descrição de como esses elementos públicos são esperados se comportarem. Apenas as assinaturas desses métodos e constantes públicos são apresentados. A implementação da classe é escondida do usuário, como se fosse uma caixa preta (Figura 7.6). Também denominamos essa situação de **encapsulamento de classe**. Dessa maneira, abstração e encapsulamento são dois lados de uma mesma moeda, de um mesmo conceito.

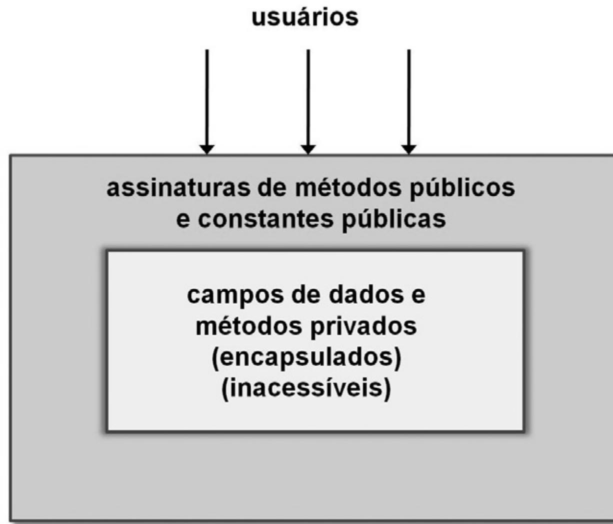


Figura 7.6 Abstração de classe, em que os usuários só enxergam a interface pública da classe.

A maneira de encapsular campos de dados e métodos é através do modificador `private`, que faz com que métodos e campos de dados sejam acessíveis apenas dentro da própria classe.

Abstrair ou encapsular uma classe não apenas simplifica o seu uso como também é útil para solucionar problemas complexos. Podemos usar abstração quando atacamos um problema complexo dividindo-o em partes, em que cada parte é uma classe que, por sua vez, é decomposta em outras classes. Nesse caso, num primeiro estágio, podemos definir apenas as assinaturas de métodos e constantes públicos dessas classes, sem se preocupar com a sua implementação concreta e exata.

Também encapsulamos uma classe por questões de segurança. Por exemplo, na versão de `Particula` do Código 7.2, algum usuário pode inadvertidamente zerar o número total de partículas (Nparticulas) fazendo p2.Nparticulas = 0. Tornando Nparticulas privada, isto é:

```
private static int Nparticulas = 0;
```

impedimos a alteração direta dessa variável. O usuário que quiser saber o número corrente de partículas continua dispondo do método público getNparticulas().

Para poder encapsular os outros campos de dados, precisamos de **métodos de get** que retornam o valor atual de um campo de dados (*e.g.* `getNparticulas()`) e de **métodos de set** que estabelecem um novo valor para a variável. Um

método de get é denominado **método de acesso** (***accessor***), e um método de set é denominado **método modificador** (***mutator***). Por convenção, usamos os prefixos `get` e `set` (sempre minúsculos) nos nomes desses métodos. Por exemplo, podemos tornar privada a variável `massa` da classe `Particula`:

```
private double massa;
```

e disponibilizar `getMassa` e `setMassa` como métodos de acesso e modificador respectivamente, isto é:

```
public double getMassa()
{
   return massa;
}
public void setMassa(double m0)
{
   massa = m0;
}
```

Código 7.4 Classe Particula com encapsulamento de campos de dados e métodos de acesso e modificador

```
public class Particula
{
   public static final double G = 9.807;  // constante aceleracao da gravidade
   private double massa = 1;
   private static int Nparticulas = 0;
   // construtores
   public Particula()
   {
      Nparticulas++;
   }
   public Particula(double m0)
   {
      massa = (m0 >= 0) ? m0 : 0;
      Nparticulas++;
   }
   // metodos modificadores e de acesso
   public double getMassa()
   {
      return massa;
   }
   public void setMassa(double m0)
   {
      massa = (m0 >= 0) ? m0 : 0;
   }
   public static int getNparticulas()
   {
      return Nparticulas;
   }
   // outros metodos publicos
   public double peso()
   {
      return massa * G;
   }
}
```

Você deve notar mais uma vantagem do encapsulamento: a facilidade de manutenção da classe. Imagine que você queira melhorar a classe `Particula` para garantir que massas jamais sejam definidas como negativas. Sendo `massa` um campo de dados privado, as alterações são todas internas à classe `Particula`, e nenhuma linha de código de programas externos precisa ser modificada. Essas alterações estão no `setMassa`:

```
public void setMassa(double m0)
{
   massa = (m0 >= 0)? m0: 0;
}
```

O Código 7.4 contém essa nova versão da classe `Particula`, e a Figura 7.7 apresenta o seu diagrama de classe UML. Neste exemplo, você deve notar que o teste de legalidade para o valor de `massa` é repetido no construtor (em vez de usar `setMassa`). Jamais chame métodos públicos da própria classe dentro de construtores, pois os métodos da classe devem ser chamados após a criação/inicialização do objeto realizada pelo construtor. A situação mais segura e pura é quando os construtores inicializam objetos com valores defaults bem definidos. Quando um construtor é chamado, primeiro todos os campos de dados são inicializados na ordem em que estão declarados (e com valores dados explicitamente ou com valores padrão `0`, `false` ou `null`) e depois o corpo do construtor é executado.

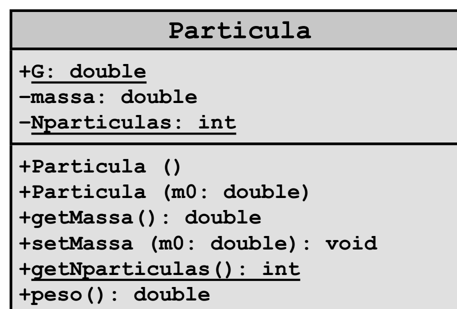


Figura 7.7 Diagrama de classe UML da classe `Particula ()`, onde + denota `public.`, - denota `private` e sublinhado denota `static`.

De agora em diante, recomendamos que você sempre defina os campos de dados como privados (`private`), exceto as constantes que possam ser úteis para os usuários. Quando não usamos nenhum dos modificadores de visibilidade (`public` e `private`), então por default as classes, os métodos e os campos de dados são acessíveis por qualquer classe do mesmo pacote (*package*).

Se os valores das propriedades de um objeto não podem ser mudados após a criação deste, dizemos que o objeto e a sua classe são **imutáveis**. Pode-

mos tornar uma classe imutável se todos os campos de dados forem `private` e não existirem métodos modificadores (set). Tornar uma classe imutável é uma importante decisão de projeto e depende da aplicação.



Nota
O fato de todos os campos de dados serem `private` e não existirem métodos modificadores não é suficiente para garantir imutabilidade. Também é necessário que não haja uma maneira indireta de mudar um campo de dados. Isso pode ocorrer quando uma classe tem, entre suas propriedades, uma outra classe que, por sua vez, tem métodos modificadores.

7.5. A referência `this`

Quando os parâmetros formais de um método têm os mesmos nomes dos campos de dados de uma classe, os campos de dados são escondidos. Nesse caso, para acessar as variáveis escondidas, devemos usar uma das seguintes práticas:

- se a variável escondida *x* é estática, então a acessamos usando *nomeDaClasse.x*;
- se a variável escondida *x* é de instância, então usamos a palavra-chave `this` como uma representante (*proxy*) do objeto que invoca o método, isto é: acessamos a variável usando `this`.*x*;

O seguinte exemplo esclarece a situação em que as variáveis são escondidas:

```
public class Qualquer
{
   private double massa;
   private static double carga;

   public Qualquer(double massa, double carga)
   {
      this.massa = massa;
      Qualquer.carga = carga;
   }
   public void setMassa(double massa)
   {
      this.massa = massa;
   }
   public static void setCarga(double carga)
   {
      Qualquer.carga = carga;
   }
}
```

Devemos sempre lembrar que o `this` representa o objeto concreto, e não a classe.

A palavra-chave `this` tem um segundo significado. Podemos usar `this`*(listaDeArgumentos)* dentro de um construtor para invocar um outro construtor da mesma classe. Por exemplo:

```
public class Particula
{
   private double massa;
   public Particula(double massa)
   {
      this.massa = massa; // mesmo caso do exemplo acima
   }
   public Particula()
   {
      this(5.0); // invoca um outro construtor coerente com a lista de argu-
      mentos
   }
}
```

É uma boa prática fazer com que um construtor sem argumentos (ou com poucos argumentos) chame um construtor com mais argumentos através de `this`*(listaDeArgumentos)*. Isso facilita a manutenção da classe. Uma restrição importante é que `this`*(listaDeArgumentos)* deve sempre ser o primeiro comando do corpo do construtor.

7.6. Herança

Um dos aspectos mais poderosos da orientação a objetos é que as classes podem participar de uma hierarquia de classes, como numa árvore de família, na qual herdam propriedades e comportamentos de seus ancestrais. Dessa maneira, derivamos novas classes a partir de classes existentes, num processo que chamamos de herança (*inheritance*). Em Java, quando uma nova classe B deriva de uma classe A, dizemos que "B estende A", e a seguinte terminologia é usada para nos referirmos a essa relação hierárquica:

- a classe original A é denominada superclasse (*superclass*) ou classe-pai (*parent class*);
- a classe derivada B é denominada subclasse (*subclasse*) ou classe-filha (*child class*).

Uma subclasse herda os campos de dados e os métodos de sua superclasse e também pode ter novos campos de dados e métodos. Por exemplo,

quadrados e triângulos podem ser definidos como subclasses de uma classe mais geral de polígonos. Uma concepção possível é que o que todos os polígonos têm em comum é um ponto particular chamado origem. Nessa concepção, o quadrado é um polígono, ou seja, `Quadrado` estende a classe `Poligono` e herda seus campos de dados e métodos, além de acrescentar suas próprias propriedades (`lado`) e métodos de instância (`getLado`, `setLado` e `area`). O mesmo pode ser feito para `TrianguloEquilatero` e `Pentagono`. A Figura 7.8 apresenta um diagrama de classes UML que modela polígonos e suas relações com essas subclasses e com uma classe agregada (`Ponto`). Na notação UML, as relações entre objetos têm representações próprias; por exemplo, herança (que é uma relação *is-a*) é indicada por uma seta, e a relação de agregação (*has-a*) é indicada por um diamante.

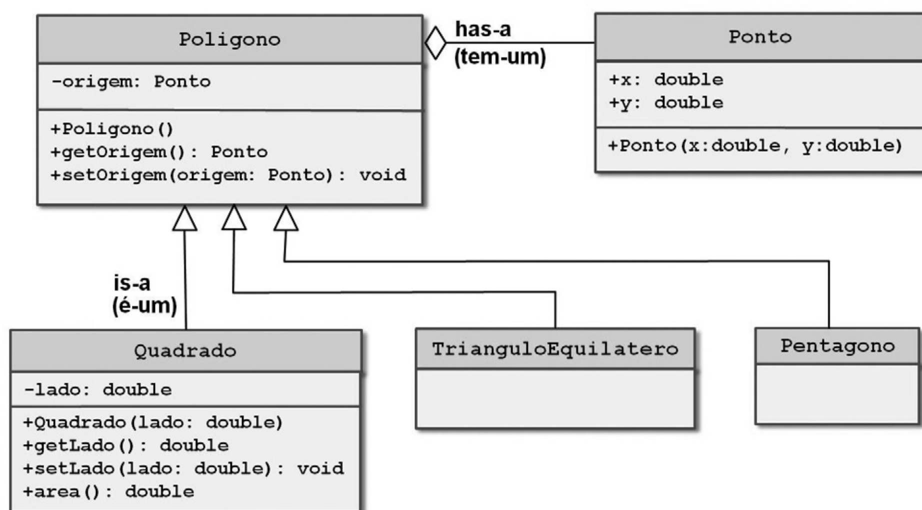


Figura 7.8 Diagrama de classe UML modelando polígonos e suas relações com outras classes. Herança é uma relação *is-a*.

Os Códigos 7.5, 7.6, 7.7 e 7.8 apresentam as implementações das classes `Poligono`, `Ponto`, `Quadrado` e `TrianguloEquilatero` da Figura 7.8. O Código 7.9 apresenta testes dessas classes e ilustra que, ao criarmos um novo Quadrado `q1`, também temos acesso ao campo de dados da classe-pai `Poligono` através do objeto `q1` (linha 6). O resultado desses testes é:

```
q1 x= 0.0 y= 0.0
q1 area= 4.0
q1 x= 3.0 y= 5.0
t1 area= 43.30127018922193
```

Código 7.5 Superclasse Poligono

```
public class Poligono
{
   private Ponto origem;
   public Poligono()
   {
      origem = new Ponto(0,0);
   }
   public Ponto getOrigem()
   {
      return origem;
   }
   public void setOrigem(Ponto origem)
   {
      this.origem = origem;
   }
}
```

Código 7.6 Classe Ponto referenciada em Poligono

```
public class Ponto
{
   public double x;
   public double y;
   public Ponto(double x, double y)
   {
      this.x = x;
      this.y = y;
   }
}
```

Código 7.7 Subclasse Quadrado que estende a classe Poligono

```
public class Quadrado extends Poligono
{
   private double lado;
   public Quadrado(double lado)
   {
      this.lado = lado;
   }
   public double getLado()
   {
      return lado;
   }
   public void setLado(double lado)
   {
      this.lado = lado;
   }
   public double area()
   {
      return lado*lado;
   }
}
```

Código 7.8 Subclasse TrianguloEquilatero que estende a classe Poligono

```
import java.lang.Math;

public class TrianguloEquilatero extends Poligono
{
   private double lado;
   public TrianguloEquilatero(double lado)
   {
      this.lado = lado;
   }
   public double getLado()
   {
      return lado;
   }
   public void setLado(double lado)
   {
      this.lado = lado;
   }
   public double area()
   {
      return lado*lado*Math.sqrt(3)/4;
   }
}
```

Código 7.9 Teste das subclasses Quadrado e TrianguloEquilatero

```
public class Main
{
   public static void main(String[] args)
   {
      Quadrado q1 = new Quadrado(2);
      System.out.println(“q1 x= ” + q1.getOrigem().x +
         ” y= ” + q1.getOrigem().y);
      System.out.println(“q1 area= ” + q1.area());
      Ponto p0 = new Ponto(3,5);
      q1.setOrigem(p0);
      System.out.println(“q1 x= ” + q1.getOrigem().x +
         ” y= ” + q1.getOrigem().y);
      TrianguloEquilatero t1 = new TrianguloEquilatero(10);
      System.out.println(“t1 area= ” + t1.area());
   }
}
```

Um esquema de herança é adequado quando faz sentido os objetos de uma subclasse serem utilizados em qualquer código que use a superclasse. Em particular, podemos atribuir um objeto de subclasse a uma variável de superclasse. O Código 7.10 ilustra essa situação, criando um vetor de polígonos (linha 5) e atribuindo um objeto quadrado de lado 10 e origem (7,15) a um desses polígonos, na linha 7. Todos os outros polígonos são genéricos e com a origem default (0,0). Outra observação importante é que as variáveis `poligono[0]` e `q2` fazem referência à mesma área de memória (linha 7). Por fim, devemos notar que uma superclasse pode ser usada como um tipo de dado (linha 5). O resultado do Código 7.10 é o seguinte:

```
poli 0 origem x= 7.0 origem y= 15.0
poli 1 origem x= 0.0 origem y= 0.0
poli 2 origem x= 0.0 origem y= 0.0
```

Código 7.10 Exemplo de superclasse usada como tipo de dado e exemplos de atribuições

```
public class Main
{
   public static void main(String[] args)
   {
      Poligono[] poligonos = new Poligono[3];
      Quadrado q2 = new Quadrado(10);
      poligonos[0] = q2;
      Ponto p2 = new Ponto(7,15);
      poligonos[0].setOrigem(p2);
      poligonos[1] = new Poligono();
      poligonos[2] = new Poligono();
      int i;
      for (i=0; i<3; i++)
      {
         System.out.println("poli "+i+" origem x= "+poligonos[i]
            .getOrigem().x + " origem y= " + poligonos[i]
            .getOrigem().y);

      }
   }
}
```

Uma subclasse também pode "passar por cima" dos métodos da superclasse, isto é: uma subclasse pode modificar a implementação de um método definido na superclasse. Chamamos essa prática de **sobreposição de método** ou **redefinição de método** (***method overriding***). Para tanto, basta o método da subclasse ter a mesma assinatura do método da superclasse. Se a assinatura for diferente, estaremos providenciando mais de um método com um mesmo nome, situação denominada **sobrecarga de método** (***method overloading***).

7.7. A Palavra-chave `super`

Subclasses herdam campos de dados e métodos de uma superclasse, mas não herdam seus construtores. Os construtores de uma superclasse só podem ser invocados a partir dos construtores de suas subclasses através do seguinte comando:

`super(`*argumentos*`)`

O comando `super` invoca o construtor de superclasse que casa com os seus argumentos. Esse comando deve aparecer na primeira linha do construtor da subclasse. Os argumentos podem estar vazios, isto é:

```
super()
```

Um outro uso da palavra-chave `super` é referenciar um método da superclasse. A sintaxe, neste caso, é a seguinte:

`super.`*método*`(`*argumentos*`);`

Normalmente não é necessário colocarmos `super` antes do método da superclasse. Entretanto, `super` é necessário quando queremos redefinir métodos (*method overriding*) e, além disso, queremos usar o método original da superclasse para completar a modificação. Por exemplo, o seguinte método de uma superclasse:

```
public double massa(double x)
{
   return 0.10*x;
}
```

pode ser redefinido (*overriding*) numa subclasse com a seguinte definição:

```
public double massa(double x)
{
   return 10 + super.massa(100*x);
}
```

Nesse caso, chamar *objetoSubclasse*`.massa(5)` retorna `60.0`.

7.8. Classes abstratas

Já vimos que abstração está associada ao grau de detalhe ou de especificação de um artefato. Quanto maior o nível de abstração, mais geral (e, portanto, menos específico) é o artefato. Superclasses sempre estão em um nível de abstração mais alto do que suas subclasses, como no exemplo de `Polígono` da Figura 7.8. Às vezes, uma superclasse se torna tão abstrata, isto é, tão geral, que não pode ter todos os seus métodos concretamente implementados. Nessas ocasiões, a superclasse se torna tão geral que acaba sendo vista mais como um modelo abstrato do que uma classe-pai com instâncias específicas compartilhadas entre classes filha. Tais classes são denominadas **classes abstratas** (***abstract classes***). Nesta seção, vamos aprender como definir e usar essas superclasses especiais.

Durante a concepção de hierarquia entre classes, notamos que alguns métodos são comuns a várias subclasses, apesar de as implementações desses

métodos dependerem de cada subclasse. Podemos notar isso na modelagem de polígonos (Figura 7.8), em que o cálculo da área de um polígono é um comportamento comum a todos os polígonos, apesar de cada polígono ter uma fórmula própria de cálculo de área. Nesse caso, declaramos `area()` como sendo um método na classe `Poligono`. Porém, esse método não pode ser implementado na classe `Poligono`, porque sua implementação depende do tipo específico de polígono. Portanto, apenas incluímos na superclasse uma assinatura de método, sem corpo, isto é, sem implementação. Esse tipo de método é denominado **método abstrato** (***abstract method***). No caso, acrescentamos a seguinte declaração na classe `Poligono`:

```
public abstract double area();
```

A inclusão de um método abstrato transforma a classe em uma classe abstrata e assim deve ser declarada, por exemplo:

```
public abstract class Poligono
{
   ...
   public abstract double area();
}
```

Os métodos abstratos são redefinidos (*overridden*) nas subclasses. Por exemplo, o método abstrato `area()` na classe `Poligono` é redefinido na subclasse `Quadrado`.

Não é possível criar instâncias de classes abstratas usando o operador `new`. Entretanto, os construtores da classe abstrata são invocados via construtores de suas subclasses. Por exemplo, um programa de teste pode conter as seguintes instruções:

```
Poligono poli1 = new Quadrado(5);
System.out.println("Area= " + poli1.area());
```

cujo resultado é:

```
Area= 25.0
```

O Código 7.11 define `Poligono` como classe abstrata, e o Código 7.12 apresenta exemplos e contraexemplos. Os códigos para `Quadrado` e `TrianguloEquilatero` continuam os mesmos (Código 7.7 e Código 7.8). O Código 7.12 também mostra o uso de um método que compara as áreas de dois polígonos. O contraexemplo (linha 22) mostra que você não pode criar uma instância a partir de uma classe abstrata usando `new` (como seria possível em superclasses concretas), porém pode usar uma classe abstrata como tipo de dado (linha 13) da mesma maneira que em superclasses concretas.

Código 7.11 Classe abstrata Poligono (novidades indicadas em negrito)

```
public abstract class Poligono
{
   private Ponto origem;
   protected Poligono()
   {
      origem = new Ponto(0,0);
   }
   public Ponto getOrigem()
   {
      return origem;
   }
   public void setOrigem(Ponto origem)
   {
      this.origem = origem;
   }
   public abstract double area();
}
```

No diagrama de classes UML, os nomes da classe abstrata e dos métodos abstratos são escritos em itálico, e o modificador `protected` é indicado por #.

Código 7.12 Exemplo de uso da classe abstrata **Poligono** (as subclasses Código 7.7 e Código 7.8 não mudam)

```
import java.lang.Math;

public class Main
{
   public static void main(String[] args)
   {
      Poligono poli0 = new Quadrado(6.5803);
      System.out.println("Area poli0= " + poli0.area());
      Poligono poli1 = new TrianguloEquilatero(10);
      System.out.println("Area poli1= " + poli1.area());
      System.out.println("Areas (-1 menor,+1 maior,0 igual):
      "+comparaArea(poli0,poli1));
      // exemplo de classe abstrata como tipo de dado
      Poligono[] poligonos = new Poligono[10];
      poligonos[0] = poli0;
      poligonos[1] = poli1;
      Quadrado q2 = new Quadrado(10);
      poligonos[2] = q2;
      Ponto p0 = new Ponto(3,5);
      poligonos[0].setOrigem(p0);
      Ponto p1 = new Ponto(7,15);
      poli1.setOrigem(p1);
      // poligonos[3] = new Poligono(); // erro: classe
         abstrata nao pode ser instanciada
      int i;
      for (i=0; i<3; i++)
      {
         System.out.println("poli " + i + " origem (x,y)= " +
            poligonos[i].getOrigem().x
               + " , " + poligonos[i].getOrigem().y + " Area= "
               + poligonos[i].area());
      }
   }
```

Código 7.12 Exemplo de uso da classe abstrata Poligono (as subclasses Código 7.7 e Código 7.8 não mudam) (cont.)

```
    public static int comparaArea(Poligono p1, Poligono p2)
    {
       final double TOL = 1.0e-2;
       if (Math.abs(p1.area()-p2.area())<TOL)
          return 0;
       else
          if (p1.area()<p2.area())
             return -1;
          else
             return +1;
    }
 }
```

Uma novidade na versão abstrata da classe `Poligono` (Código 7.11) é o uso do modificador de visibilidade (ou acessibilidade) `protected`, que tem o mesmo nível de acessibilidade do `public`, exceto pelo fato de não permitir acessos a partir de um outro pacote (*package*). Quando escrevemos uma classe abstrata queremos garantir que apenas os extensores dessa classe tenham acesso a ela (por isso a restringimos ao universo do pacote em que foi definida, usando o modificador `protected`).

Numa escala de força de acessibilidade temos `private` como a mais fraca (*i.e.* a mais restritiva) e `public` como a mais forte: `private` – `default` – `protected` – `public`. Uma subclasse pode redefinir (*override*) um método `protected` na sua superclasse e tornar sua visibilidade `public`. Entretanto, uma subclasse nunca pode enfraquecer o nível de acessibilidade de um método definido na sua superclasse. Portanto, se um método é `public` na superclasse, ele deve ser definido como `public` na subclasse.

Em Java, há bastante liberdade na definição de classes e métodos abstratos. Por exemplo, uma subclasse pode ser abstrata mesmo que sua superclasse não seja. Uma subclasse pode até mesmo redefinir um método concreto de uma superclasse concreta declarando-o abstrato, mas, em geral, só praticamos isso quando a implementação do método da superclasse se torna inválido na subclasse. Um outro exemplo de liberdade na definição de classes abstratas é o caso de declarar uma classe como sendo abstrata mesmo que ela não contenha métodos abstratos. De uma maneira geral, sempre tentamos definir o maior número possível de métodos concretos em uma classe abstrata, pois isso aumenta a funcionalidade da classe abstrata. Outras vezes definimos classes abstratas que não contêm métodos concretos. Um exemplo, neste livro, é a classe abstrata `GameObject` (objeto de jogo).

7.9. As classes GameObject e Player

No **javaPlay**, o motor de jogos especialmente desenvolvido para este livro, todos os elementos do jogo derivam da classe abstrata `GameObject`, tais como o jogador (`Player`), o placar de vidas (`Score`) e os inimigos:

```
public abstract class GameObject
{
   public int x;
   public int y;

   public abstract void step(long timeElapsed);
   public abstract void draw(Graphics g);
}
```

O sistema de coordenadas no Java está ilustrado na Figura 7.3.

O coração de um motor de jogo é um tipo de laço (chamado de *game loop*) que se repete a cada passo de tempo. Na classe `step` deve ser implementado o que ocorre em cada passo de tempo, e em `draw` são definidos os aspectos gráficos do `GameObject`. No Capítulo 8 deste livro, os aspectos gráficos são tratados com detalhes, junto com outras opções de atualizar cada passo de tempo.

Mesmo sem termos apresentado os detalhes de como são considerados a biblioteca gráfica e os *sprites* (pequenos retângulos de imagem) no Java, podemos entender a implementação de um objeto concreto de jogo. Tipicamente, os jogos 2D trabalham apenas com *sprites,* e o `Player` é um pequeno *sprite* animado. O Código 7.13 é uma versão muito simples e pouco usual de jogo que implementa um jogador (`Player`) composto por um quadrado e duas paredes. Nesse objeto composto, o quadrado fica sendo continuamente rebatido pelas duas paredes (Figura 7.9). Para rodar esse "jogo", você precisa carregar a **Versão Nível Mini** do javaPlayer disponível na página Web deste livro. A Versão Nível Mini tem apenas dois pacotes, com as seguintes classes:

Pacote demo:

- Main.java
- Player.java

Pacote javaPlay:

- GameCanvas.java
- GameEngine.java
- GameObject.java

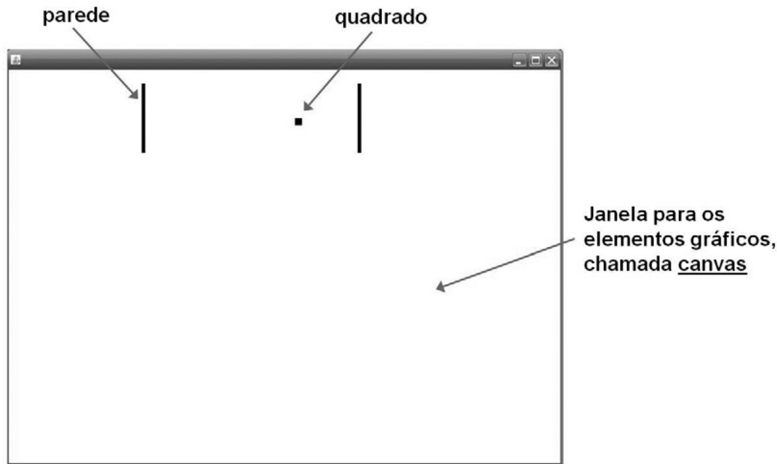


Figura 7.9 Objeto de jogo composto por um quadrado e duas paredes. O motor de jogo javaPlay na sua Versão Nível Mini faz com que o quadrado fique num vai e vem horizontal entre as duas paredes.

Código 7.13 Classe Player que estende GameObject e implementa o comportamento da Figura 7.9

```
import java.awt.Graphics;    // classe grafica do Java
import javaPlay.GameEngine;
import javaPlay.GameObject;

public class Player  extends GameObject
{
   private int delta;  // deslocamento em pixels
   private int min;    // coordenada limite inferior (em pixels)
   private int max;   // coordenada limite superior (em pixels)

   public Player()
   {
      // coordenadas onde nasce o quadrado
      x = 200;
      y = 100;
      // propriedades inicializadas
      delta = 1;
      min = x;
      max = 500;
   }

   public void step(long timeElapsed)
   {
      // atualizacao a cada passo de tempo que faz o vai-e-vem
         do quadrado
      x += delta;
      if(x > max)
         delta *= -1;
      if(x < min)
         delta *= -1;
   }

   public void draw(Graphics g)
   {
      // canvas branco (white)
```

Código 7.13 Classe Player que estende GameObject e implementa o comportamento da Figura 7.9 (cont.)

```
            g.setColor(Color.WHITE);
            g.fillRect(0, 0, GameEngine.getInstance().getGameCanvas().
               getWidth(),
                  GameEngine.getInstance().getGameCanvas()
                     getHeight());
            // desenha quadrado preto (black)
            g.setColor(Color.BLACK);
            g.fillRect(x, y, 10, 10);
            // desenha paredes
            g.fillRect(195, 50, 5, 100);
            g.fillRect(505, 50, 5, 100);
         }
      }
```

EXERCÍCIOS

E7.1 Projete e teste uma classe `Circulo` simples e totalmente pública cujas propriedades são raio, coordenada x0 do centro e coordenada y0 do centro. Defina dois construtores, sendo um deles sem argumentos, e métodos para calcular área e perímetro. Entre as atividades do teste, altere as propriedades. Desenhe, à mão, num papel, o diagrama de classe UML.

E7.2 Altere a classe `Circulo` do Exercício E7.1 para usar uma classe `Ponto2D` totalmente pública que contenha apenas as coordenadas x e y e dois construtores, um deles sem argumentos e com os valores default x = 0 e y = 0.

E7.3 Defina e teste uma classe `Circulo` que contenha os seguintes elementos estáticos: a constante PI = 3.1416, uma variável que vai acumulando o número de círculos criados e um método que consulta o valor desse acumulador.

E7.4 Reescreva todas as classes dos Exercícios anteriores (E7.1 a E7.3) usando, onde for adequado, a palavra-chave `this` e encapsulando os dados. Desenhe, à mão, num papel, os diagramas de classe UML.

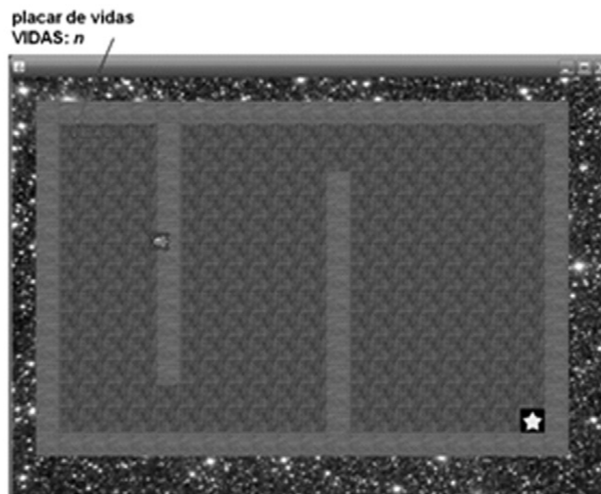
E7.5 Projete e teste uma classe `Quadrilatero`, usando `this` e encapsulando dados, tendo como propriedades a origem e o vetor de Pontos2D que define a geometria do quadrilátero. Acrescente os comportamentos que achar mais adequados.

E7.6 Modele, desenhe diagramas de classe e teste uma classe abstrata `Poli2D` de polígonos 2D, com construtores `protected`, que tenha como subclasses: `Quadrilatero` e `Triangulo`. Essas subclasses, por sua vez, têm subclasses mais simples, como `Quadrado` e `TriangEquilatero`. Teste criando vetores de objetos (que usa superclasses como tipos de dados).

E7.7 Altere a **Versao Nível Mini** do motor **javaPlay** (ver Seção 7.9) para que haja dois quadrados animados e defasados. O código original da Versão Nível Mini está na página Web do livro.

E7.8 Altere a **Versão Nível 00** do motor **javaPlay**, que está na página Web do livro, de maneira que haja um placar de vidas decrementando toda vez que o *Player*

cruzar determinadas regiões do canvas (por exemplo, as regiões que correspondem às paredes internas). Para tanto, crie uma classe `Score` como estendendo `GameObject` e teste, a cada `step`, se o *Player* entra em uma das regiões (são testes simples de coordenadas; você não precisa implementar um esquema de colisão entre *sprites*). Mesmo sem ter pleno conhecimento sobre os detalhes da parte gráfica, você pode resolver este exercício analisando a versão mais completa do motor *javaPlay* (**Versão Nível 0**), que tem um placar de vidas por colisão com inimigos. Aprendemos a programar jogos exatamente desta maneira: analisando e adaptando códigos mais complexos, numa prática de abstração (isto é: usando as funcionalidades dos artefatos, sem precisar entender detalhes de suas implementações). Você pode fazer variantes desse exercício criando regiões ocultas que aumentam o placar de vidas.



Capítulo

8

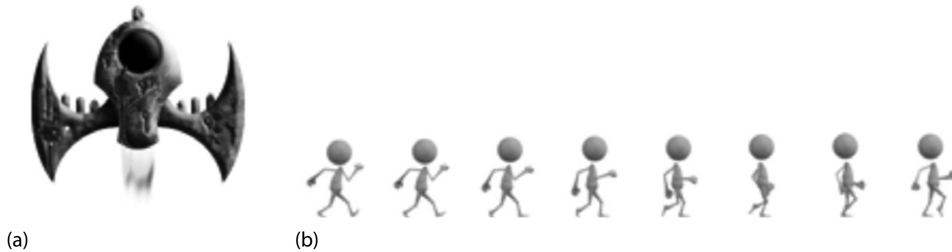
Programação Gráfica

Até este capítulo, ainda não foi possível desenvolver nenhum jogo completo. A razão para isso é que estão faltando formas de se lidar com um tipo de elemento que é fundamental para qualquer jogo ou aplicação multimídia: elementos gráficos.

Mais ainda, para fazer jogos tridimensionais é necessário usar modelos geométricos 3D, feitos em softwares de modelagem, tais como o 3DS MAX, Maya ou Blender. As placas gráficas e os motores se encarregarão de gerar imagens para esses modelos, usando algoritmos que calculam iluminação, projeção de texturas etc. Jogos 3D apenas se tornaram viáveis quando surgiram as placas gráficas, uma vez que é necessário que haja um hardware dedicado apenas a calcular polígonos e desenhá-los na tela. Antes da existência das placas gráficas já existiam videogames, mas tinham a restrição de não poderem ser tridimensionais. Embora as placas gráficas já estejam dominando o mundo – e há quem diga que as placas gráficas são mais poderosas que as próprias CPUs – para o escopo deste livro estaremos lidando apenas com jogos 2D.

Para os jogos bidimensionais usam-se basicamente imagens: imagens como fundo, imagens como personagens, imagens como elementos de interação etc.

Para jogos 2D existe um conceito muito importante e fundamental para sua concepção: os *sprites* – retângulos que possuem uma textura, uma imagem ou até mesmo um vídeo projetados sobre si. Além disso, é possível que esse *sprite* possua uma região transparente, permitindo que seu contorno não seja necessariamente retangular. A Figura 8.1 ilustra alguns exemplos de *sprites*.



(a) (b)

Figura 8.1 Exemplos de *sprites*. Enquanto (a) é uma imagem simples, (b) representa um *sprite* composto, que poderá ser usado para uma animação.

Um jogo 2D basicamente consiste em manipular, mover, mostrar, esconder, bater, matar *sprites*... Outro conceito importante em jogos 2D é o do plano de fundo: uma imagem que será colocada por detrás de tudo. Essa imagem pode ficar parada o tempo todo ou pode mover-se, dando a impressão de que a câmera é que está se movendo.

O conceito de um *sprite* se encaixa perfeitamente no de uma classe: um jogo poderá conter uma série de objetos gráficos, sendo cada um representado por uma imagem, com atributos, tais como posição, tamanho e rotação, que podem ser atualizados constantemente. Esta seção apresenta o modo gráfico do Java. Para o desenvolvimento das aplicações estará sendo usado o javaPlay, sendo que nas Seções 8.1 e 8.2 será usada uma versão resumida do motor, conforme você poderá conferir nos respectivos códigos-fonte.

8.1. As classes *Graphics* e *Color*

Na nossa vida, constantemente estamos precisando de diversos tipos de serviços: um médico que seja capaz de tratar uma doença, um mecânico que concerte o nosso carro ou um professor que nos ensine algo desconhecido. A programação baseada em objetos funciona mais ou menos assim: os programas requerem funcionalidades de natureza muito diferentes entre si. Para tanto, “contrataremos” o serviço de algumas classes que saibam como lidar com um problema específico.

A classe *Graphics* é uma classe capaz de executar e lidar com tarefas gráficas básicas, e será através dela que manipularemos os elementos gráficos mais elementares. Ressaltamos que ela é uma classe básica; portanto, caso seja necessário algum tipo de tarefa mais complexa (tal como lidar com modelos 3D, calcular iluminação ou gerenciar placas gráficas), serão necessárias outras classes mais sofisticadas – sendo muitas delas derivadas da classe *Graphics*. Outra informação importante sobre essa classe é que ela é do tipo *abstract*.

Diremos que a classe *Graphics* toma conta do contexto gráfico. Isso quer dizer que será através dela que executaremos tarefas gráficas, tal como desenhar um círculo, um retângulo ou até mesmo uma imagem ou *sprite* na tela. Podemos imaginar que essa classe será semelhante a uma caneta: sempre que mandarmos alguém desenhar alguma coisa, precisaremos entregar a ele/a a caneta, como instrumento. Outro aspecto muito importante da classe *Graphics* é que ela se responsabilizará por abstrair o sistema operacional ou até mesmo o dispositivo gráfico do usuário. Vale lembrar que cada sistema operacional gerencia de forma diferente o seu modo gráfico. Portanto, se fossemos implementar as tarefas que a classe *Graphics* faz na "força bruta", teríamos de programar acessos para cada plataforma específica. Através dela, podemos lidar com a tela gráfica, ao qual chamaremos de *canvas*, sem nos preocuparmos quais são as suas capacidades ou o modo gráfico do usuário final. Na arquitetura básica do javaPlay, todo *game object* tem um método, chamado *draw*, que é responsável por desenhá-lo na tela. Como o objeto *Graphics* é necessário para qualquer tarefa de desenho, esse método precisa receber como argumento o objeto *Graphics*, como pode ser visto no esqueleto da definição de uma classe qualquer de um *game object*, no Código 8.1.

Código 8.1 Esqueleto de um game object, evidenciando a presença do objeto *Graphics*

```
public class elementoQualquer extends GameObject
{
   public elementoQualquer()
   {
   }
   public void step(long timeElapsed)
   {
   }
   public void draw(Graphics g)
   {
   }
}
```

É importante notar que num jogo haverá muitos *game objects*. Entretanto, todos os métodos *draw* receberão o mesmo objeto *g* como parâmetro, uma vez que todos estão desenhando no mesmo contexto. Embora o contexto não seja equivalente a uma tela, podemos fazer uma analogia e dizer que todos os *game objects* estão desenhando na mesma tela.

Antes de vermos mais a fundo como funciona um objeto *graphics*, é necessário apresentar uma classe muito próxima, que é a classe *Color*. Não que sejam próximos por serem parecidos, mas sim por serem "amigos". Isso quer dizer que quando a classe *Graphics* estiver presente, é comum termos de chamar também instâncias da classe *Color.* Essa dependência é mais ou menos natural: para usar uma caneta ou um pincel sobre um papel, naturalmente essa caneta deverá ter uma cor específica. Fazendo a analogia, se a classe *Graphics* tiver de desenhar elementos na tela, é necessário que antes sejam especificadas as cores com as quais esses elementos serão desenhados.

Quando um artista vai pintar um quadro, ele não pode prever todas as cores de que precisará para a sua obra. Ele irá adquirir algumas tintas básicas e a partir delas fará combinações de forma a obter as cores que forem sendo necessárias. Em computação ocorre algo semelhante: para compor qualquer cor, usaremos as cores primárias, que no caso da eletrônica são o Vermelho, Verde e Azul. Quando descrevemos uma cor seguindo esse padrão, dizemos que estamos usando o modelo RGB – *Red*, *Green*, *Blue*. Assim sendo, um objeto do tipo cor terá basicamente três atributos óbvios: o seu valor de R, G e B. Em geral, cada um desses atributos será do tipo *integer*, que poderá variar entre os valores de 0 a 255. A seguir, alguns exemplos de cores em RGB:

(0, 0, 0)	: Preto
(255, 0, 0)	: Vermelho "puro"
(0, 255, 0)	: Verde "puro"
(0, 0, 255)	: Azul "puro"
(255, 255, 255)	: Branco
(128, 128, 128)	: Cinza em meio tom
(255, 255, 0)	: Amarelo
(255, 0, 255)	: Magenta
(0, 255, 255)	: Ciano
(255, 200, 0)	: Laranja

É comum na computação gráfica que em alguns casos se usem valores reais em vez de inteiros, para representar o intervalo de cores. Nesse caso, o intervalo para cada uma pode variar de 0 a 1, sendo os atributos R, G e B do tipo *float*. Usando esse modelo, o vermelho seria (1.0, 0.0, 0.0), e o cinza meio-tom seria (0.5, 0.5, 0.5). Em Java é possível usar qualquer um dos modelos, mas é importante adotar uma convenção na aplicação para evitar confusões óbvias. Neste livro estamos usando o tipo *integer*. Veja no Código 8.2 como se pode criar e realizar algumas operações básicas através da classe *Color*:

Código 8.2 Exemplos de uso da Classe *Color*

```
1    Color minhaCor;
2    Color outraCor;
3    Color maisUmaCor;
4    int vermelho = 100;
5    int verde = 20;
6    int azul = 100;
7    int outroAzul;
8    minhaCor = new Color (vermelho, verde, azul);
9    outroAzul = minhaCor.getBlue ();
10   outraCor = new Color (0, 0, outroAzul);
11   maisUmaCor = Color.blue;
```

Observando o exemplo, ficam evidentes os principais atributos da classe, que são as três cores primárias usadas para a construção de uma cor qualquer. A classe *Color* é um pouco mais sofisticada do que parece, porém para o escopo deste curso não é necessário ver todos os atributos e métodos que estão presentes. Apenas ressaltamos que, além do construtor, existem métodos que permitem ler qual é a cor de um objeto desse tipo. A linha 9 do Código 8.2 mostra o método *getBlue*(), que retornará o valor do atributo blue. De igual maneira, existem os métodos *getRed*() e *getGreen*(). A linha 11 atribui a um objeto do tipo *Color* um valor de cor predefinido – no caso, azul. Para tanto, usam-se uma das constantes *Colors* predefinidas. Não é necessário criar um novo objeto *Color* para essa tarefa, pois as constantes *Color* são *static* e são definidas uma única vez quando a classe *Color* é carregada na memória durante a execução. Existem constantes para as cores mais comuns, tais como Color.green, Color.red, Color.yellow, Color.white, Color.pink, Color.black etc.

A classe *Graphics* constantemente fará uso de objetos do tipo *Color,* pois sempre que for executado algum método de desenho, será necessário dizer qual é a cor daquilo que se deseja pintar ou desenhar. Agora que a classe *Color* já foi esmiuçada, podemos entrar em detalhes na classe *Graphics*.

A classe *Graphics* basicamente consiste num executor de tarefas gráficas. Não será usual termos de instanciar essa classe. No caso deste livro, o engine fica a cargo da tarefa. A área de atuação da classe é a tela do computador, que para a aplicação em questão será chamada de Canvas e no javaPlay é chamado de GameCanvas. É importante ter em conta que a tela é descrita por uma matriz de pontos (*pixels*), sendo o canto superior esquerdo a origem das coordenadas, tal como pode ser visto na Figura 8.2. O canto inferior direito corresponde ao limite da tela e será dado pela coordenada (largura, altura). Como uma aplicação pode ser executada em diversos computadores diferentes, os valores de largura e altura podem ser totalmente diferentes. Assim, em vez de trabalharmos com limites absolutos, deixaremos a cargo da aplicação para que verifique quais são esses valores. Qualquer pixel da tela deverá ser endereçado por um par de coordenadas (x, y). Como é óbvio, esses valores não podem ser negativos ou maiores que os limites estabelecidos pela resolução da aplicação. Caso a aplicação esteja rodando numa janela, as coordenadas referem-se ao canto superior esquerdo e inferior direito da janela, sendo a largura e altura valores referentes ao tamanho da janela e não da tela.

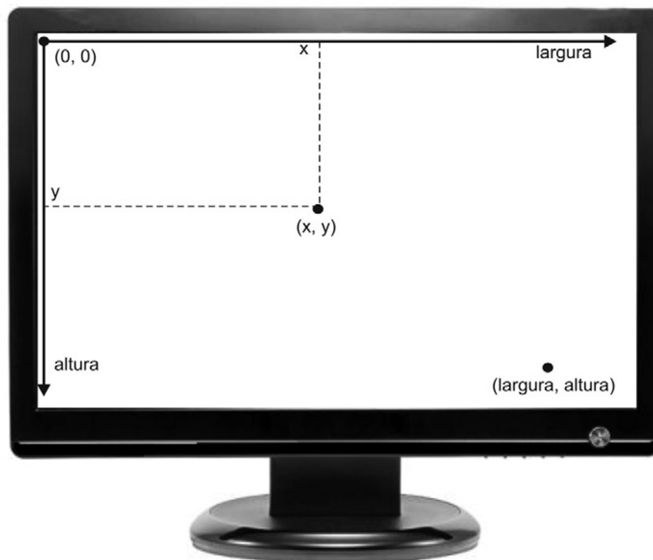


Figura 8.2 Sistemas de coordenadas do JAVA.

Vejamos agora alguns métodos mais usuais da classe *Graphics*, tendo em conta que *g* é um objeto desse tipo.

g.setColor (Color c)
Diz qual será a cor ativa. A partir deste ponto, enquanto não for especificada outra cor, tudo o que for desenhado terá a cor estipulada por c.

g.fillRect (int x, int y, int largura, int altura)
Preenche um retângulo da tela, descrito pela coordenada superior esquerda (x, y) e com a respectiva largura e altura. Esse preenchimento é feito usando-se a última cor que foi ativada, pelo método setColor().

g.drawRect (int x, int y, int largura, int altura)
Desenha um retângulo, porém neste caso apenas desenha a borda, não pintando o interior.

g.drawLine (int x1, int y1, int x2, int y2)
Desenha uma linha partindo da coordenada (x1, y1) até a coordenada (x2, y2).

drawOval(int x, int y, int largura, int altura)
Desenha uma figura oval colocada na coordenada (x, y) e com a respectiva largura e altura. Vale ressaltar que se estas medidas forem iguais, a figura oval será na verdade uma circunferência.

g.fillOval (int x, int y, int largura, int altura)
Equivalente ao g.DrawOval, porém pinta o interior do oval.

g.clearRect (int x, int y, int largura, int altura)
Limpa uma área da tela, definida pelo canto superior esquerdo (x, y) e com a respectiva largura e altura.

O trecho de Código 8.3 exemplifica o uso de *Graphics*, desenhando uma tela do Pong, que está na Figura 8.3.

Código 8.3 Exemplo do uso da classe Graphics para plotar uma tela do Pong

```
g.setColor(Color.black);
g.fillRect(0,0,800,600);

// Simulando os velhos monitores de fósforo verde...
g.setColor(Color.green);
g.fillRect(15,200, 15, 100);
g.fillRect(770,100, 15, 100);
g.fillOval(400,200,20,20);
```

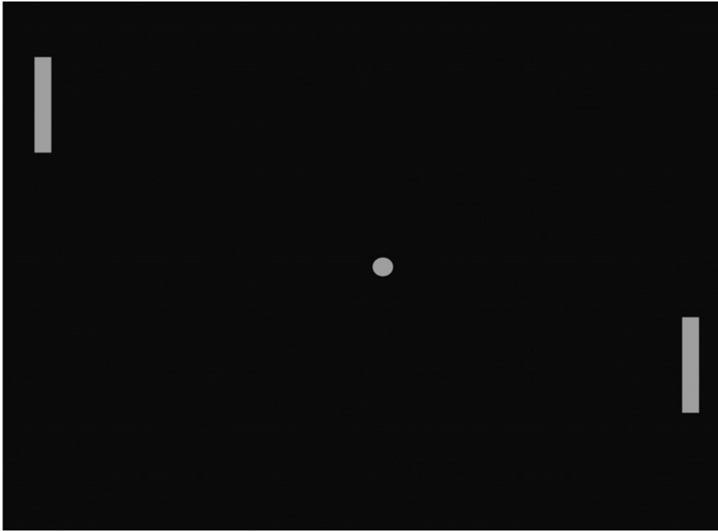


Figura 8.3 Imagem produzida pelo trecho de Código 8.3. (Ok, no Pong original, a bolinha era quadrada, mas vamos colocar circular para podermos experimentar mais recursos da classe *Graphics*...)

A linha 2 do Código 8.3 pinta um retângulo preto, usando as coordenadas 0,0 e 800, 600. Nesse caso, estamos imaginando que a tela do jogador tem a largura e altura de 800 e 600 respectivamente. Entretanto, assumir esses valores pode desagradar alguém que comprou um monitor de 26 polegadas em alta resolução... Na verdade, é impossível saber *a priori* qual é a resolução da tela do usuário final. Assim, deveremos substituir o valor absoluto de largura e altura por um valor que será consultado. A classe principal do javaPlay possui um método capaz de instanciar o canvas que está sendo usado por ele quando bem se desejar. Esse método é chamado da seguinte forma:

GameEngine.getInstance().getGameCanvas()

O GameCanvas, que pode ser entendido como a tela do game, possui, por sua vez, um método capaz de ler o valor da resolução de largura e outro para a altura da tela do usuário: getWidth() e getHeigh(). Assim sendo, ao realizar as seguintes chamadas:

int largura = GameEngine.getInstance().getGameCanvas().getWidth();

int altura = GameEngine.getInstance().getGameCanvas().getHeight();

teremos que as variáveis largura e altura receberão a resolução da tela que o jogador possui, que pode variar desde uma tela de muitíssimas polegadas com alta resolução até uma pequena tela de um celular. Com esse recurso, o Código 8.4 torna-se muito mais elegante que o 8.3, possibilitando que o jogador possa escolher livremente a sua resolução de tela.

Código 8.4 Tela do Pong, agora independente de resolução

```
int largura=GameEngine.getInstance().getGameCanvas().getWidth();
int altura=GameEngine.getInstance().getGameCanvas().getHeight();

g.setColor(Color.black);
g.fillRect(0,0,largura,altura);

g.setColor(Color.green);
g.fillRect(15,200, 15, 100);
g.fillRect(largura - 30,100, 15, 100);

// Agora a bolinha quadrada, como nos velhos tempos…
g.fillRect(400,200,20,20);
```

Até este momento apenas vimos trechos de código para exemplificar o modo gráfico. Vamos ver agora como integrar isso dentro do javaPlay em si e começar a esboçar a primeira tarefa do jogo Pong, que consiste no movimento da bolinha de um lado para o outro da tela.

Vimos que a arquitetura do javaPlay consiste em criar uma série de *game objects*. Dentro da função main() adicionamos esses objetos na estrutura do jogo em si, ficando a implementação do jogo a cargo da definição do game object e da programação dos seus métodos step() e draw().

Inicialmente, vamos implementar a aplicação inteira dentro de um único *game object*, ao qual chamaremos de jogo. Os métodos step() e draw() do objeto jogo irão atualizar a posição de cada elemento dinâmico, que neste caso será apenas a bolinha. Posteriormente, quando formos criar o jogo por completo, iremos melhorar essa arquitetura, criando um *game object* para cada elemento dinâmico e implementando o desenho de cada um separadamente. Na primeira implementação apenas nos preocuparemos com o movimento da bolinha, sendo que desenharemos as barrinhas apenas para fins "estéticos". O Código 8.5 implementa os atributos e o construtor básico do *game object* jogo.

Código 8.5 game object que implementa o movimento da bolinha

```

import java.awt.Color;
import java.awt.Graphics;
import javaPlay.GameEngine;
import javaPlay.GameObject;

public class Jogo extends GameObject
{
   private int posBolaX; // posição X e Y da bolinha em cada instante
   private int posBolaY;
   private int altura;    // altura e largura da tela
   private int largura;
   private int velBolaX;  // Velocidade, em Pixels, da bolinha
```

Código 8.5 game object que implementa o movimento da bolinha (cont.)

```
14       private int velBolaY;
15
16       public Jogo()
17       {
18          altura = GameEngine.getInstance().getGameCanvas().getHeight();
19          largura = GameEngine.getInstance().getGameCanvas().getWidth();
20
21          // Inicializa-se a bolinha no meio da tela
22          posBolaX = largura/2;
23          posBolaY = altura/2;
24
25          // Velocidade inicial da bolinha em cada eixo, em pixels
26          velBolaX = 5;
27          velBolaY = 3;
28       }
29    }
```

Para esse Game Object é necessário incluir as classes GameEngine e GameObject, de modo que se possa usar o código do javaPlay disponibilizado. Como estaremos executando tarefas gráficas, é necessário incluir os inseparáveis *Graphics* e *Color*, como pode ser visto pelas linhas 2 e 3. No campo de atributos, criaremos variáveis que descrevem a posição da bolinha, bem como sua velocidade. Ambos descrevem valores em pixels: a coordenada da tela da bolinha bem como o número de pixels que a bolinha se desloca em cada game loop. Adicionalmente, colocaremos como atributos do jogo a resolução da tela do jogador. Entretanto, é bom deixar claro que esse não é o melhor lugar para colocá-los, já que não são atributos da bolinha, mas do jogo completo.

O construtor dessa classe fará a leitura da resolução da tela (linhas 18 e 19) e colocará a bolinha numa posição qualquer da tela (linhas 22 e 23), sendo que neste exemplo estamos colocando-a exatamente no centro. Finalmente cria-se um valor para a velocidade da bolinha em cada eixo (linhas 26 e 27). Perceba que, ao colocar velocidades diferentes para cada eixo, a bolinha andará mais num sentido do que em outro.

Código 8.6 Atualização lógica da bolinha para cada frame

```
public void step(long timeElapsed)
   {
      posBolaX += velBolaX;
      posBolaY += velBolaY;

      if (posBolaX > largura) velBolaX *= -1;
      if (posBolaX < 0) velBolaX *= -1;

      if (posBolaY > altura) velBolaY *= -1;
      if (posBolaY < 0) velBolaY *= -1;

   }
```

O Código 8.6 implementa a lógica de movimento da bolinha. Como foi visto, cada *game object* possui um método step() que é responsável por fazer sua atualização lógica. Enquanto ainda não colocarmos as barras dos jogadores, a lógica da bolinha "quicante" consiste em dar um passo na direção *x* e outro na direção *y* (linhas 3 e 4). Em seguida, devemos verificar se a bolinha chocou-se com as bordas da tela (linhas 6, 7, 9 e 10). Se isso ocorrer, basta inverter a direção da velocidade de movimento da bolinha. Para tanto, invertemos o sinal da velocidade, fazendo com que ela mude de sentido.

Código 8.7 Desenho do jogo a cada passo da aplicação

```
1    public void draw(Graphics g)
2    {
3       g.setColor(Color.black);
4       g.fillRect(0, 0, largura, altura);
5
6       g.setColor(Color.green);
7       g.fillOval(posBolaX, posBolaY, 20, 20);
8
9       g.fillRect(15, 200, 15, 100);
10      g.fillRect(largura - 30,  100, 15, 100);
11   }
```

Após executar o método de step() de cada *game object*, o motor irá executar o método de draw() para cada um. No caso desta aplicação, como jogo é o único *game object*, o seu método draw() se responsabilizará por desenhar tudo. Primeiramente limpa-se a tela por inteiro (linhas 3 e 4), em seguida desenha-se a bolinha, na posição já atualizada pelo método step() e finalmente, apenas por razões estéticas, desenham-se as barrinhas, que neste exemplo ficam sempre na mesma posição.

Finalmente, o Código 8.8 mostra o código referente à classe principal do javaPlay, em que instancia-se o objeto jogo (linha 6), adiciona-se o mesmo ao projeto (linha 7) e finalmente executa-se o motor propriamente dito (linha 8).

Código 8.8 Classe principal do engine

```
1    import javaPlay.GameEngine;
2    public class Main
3    {
4       public static void main(String[] args)
5       {
6          Jogo jogo = new Jogo();
7          GameEngine.getInstance().addGameObject(jogo);
8          GameEngine.getInstance().run();
9       }
10   }
11
```

Ao executar esse programa, você perceberá um pequeno problema, provavelmente indesejado, para o futuro jogo: ao chocar-se na borda direita e na borda de baixo da tela, a bolinha momentaneamente desaparece. Isso ocorre por uma razão muito simples: a posição que descreve a bolinha refere-se ao canto superior esquerdo desta, devido à forma como o método g.fillOval() é implementado. Assim, apenas é detectado que houve de fato uma colisão quando toda a bolinha já saiu da tela e esse canto tocou a borda. Para corrigir o problema, devemos fazer o teste de colisão somando-se o tamanho da bolinha em cada um dos eixos, tal como pode ser visto nas linhas 6 e 9. Neste caso, estamos considerando que a altura e largura da bolinha são de 20 e 20 respectivamente, mas futuramente poderemos ter de usar valores parametrizados.

Código 8.9 Correção da colisão da bolinha com as bordas

```
1   public void step(long timeElapsed)
2      {
3         posBolaX += velBolaX;
4         posBolaY += velBolaY;
5
6         if (posBolaX + 20 > largura) velBolaX *= -1;
7         if (posBolaX < 0) velBolaX *= -1;
8
9         if (posBolaY + 20 > altura) velBolaY *= -1;
10        if (posBolaY < 0) velBolaY *= -1;
11
12     }
```

8.2. Escrevendo texto no modo gráfico

Quando estamos trabalhando no modo gráfico, a área de "escrita" é como um papel, em que tudo precisa ser desenhado, inclusive se formos escrever um texto. Dessa forma, caso seja usado um comando do tipo System.out.print("Hello World!") ocorrerá uma incompatibilidade de formatos, uma vez que esse comando enviará algo no modo texto para uma janela (*canvas*) do tipo gráfico. Para resolver o impasse, o objeto *Graphics* possui métodos próprios para escrita. Esse processo fará com que o texto escrito seja praticamente como outros objetos gráficos quaisquer, podendo inclusive configurar sua cor, tamanho e posição na tela.

Antes de vermos como a classe *Graphics* trata a escrita de texto, será necessário apresentar uma nova classe, chamada de *Font*. Da mesma forma que foi necessário configurar qual seria a cor usada antes de desenhar algo na tela, também é necessário dizer como será a fonte que será usada, ao se escre-

ver algo na tela. Essas propriedades referem-se basicamente ao tipo da fonte, estilo e tamanho.

Embora haja muitos métodos e atributos para essa classe, os métodos mais usados são:

public Font (String fonte, int style, int size)

Cria um objeto do tipo *Font*, com a fonte especificada pela string fonte, estilo especificado pelo inteiro style e tamanho pelo inteiro size. O estilo de uma fonte pode ser PLAIN (normal), BOLD (negrito) e ITALIC (itálico). Para tanto, existem três constantes que podem ser usadas para referência: Font.PLAIN, Font.BOLD e Font.ITALIC, respectivamente.

public string getName()

Retorna a fonte de um objeto tipo *font*.

public int getStyle()

Retorna o estilo de um objeto tipo *font*.

public int getSize()

Retorna o tamanho de um objeto tipo *font*.

O processo para escrever, estando no modo gráfico, consiste nos seguintes passos: configurar a cor da fonte, configurar a fonte a ser usada e finalmente realizar a operação de escrita. Para configurar a fonte e realizar a escrita propriamente é necessário usar dois métodos que pertencem à classe *Graphics*:

g.setFont(aFonte)

Estipula o objeto aFonte como sendo a fonte que servirá para realizar a escrita, a partir deste momento.

g.drawString("texto no modo String", posicaoX, posicaoY)

Escreve, no modo gráfico, a string passada como parâmetro, sendo a borda superior esquerda da área de escrita estipulada pela coordenada posicaoX, posicaoY.

O Código 8.10 implementa um exemplo de escrita ao método draw() anterior, imprimindo um texto fixo e um contador variável, que é o número de vezes que a bolinha bateu em cada borda da tela.

Código 8.10 Exemplo de escrita no modo gráfico

```
1   public void draw(Graphics g)
2   {
3      g.setColor(Color.black);
4      g.fillRect(0, 0, largura, altura);
5
6      g.setColor(Color.green);
7      g.fillOval(posBolaX, posBolaY, 20, 20);
8
9      g.fillRect(15, 200, 15, 100);
10     g.fillRect(largura - 30,  100, 15, 100);
11
12     Font aFonte = new Font (“serif”, Font.BOLD, 12);
13     g.setFont(aFonte);
14     g.drawString(“batidas”, 50, 50);
15     g.setFont(new Font (“serif”, Font.ITALIC, 14));
16     g.drawString(Integer.toString (contador), 150, 50);
17  }
```

Para o código em questão é necessário criar dois atributos para a classe Jogo, que são o contador e o objeto aFonte. O contador precisa ser incrementado a cada vez que o sentido da bolinha é invertido. A linha 12 cria uma especificação de uma fonte através do método de criação da classe Fonte. Na linha 13 especifica-se qual será a fonte que a partir desse momento será usada e em seguida escreve-se a palavra “batidas” na posição 50, 50 da tela. Observe que a cor da escrita será verde, pois não se informou até o momento que ocorreria alguma troca de cor, portanto, continua valendo a cor especificada na linha 6. Após a primeira escrita, será feita uma troca da configuração da fonte. Porém, neste caso, a linha 15 ilustra um atalho para esse processo, uma vez que será criada uma fonte sem a necessidade da criação de um objeto intermediário, passando-se diretamente ao parâmetro fonte do método setFont um criador de um objeto. Finalmente, na linha 16 escreve-se na tela o contador de batidas. Nesse caso, há um pequeno detalhe: o contador é do tipo inteiro, e o parâmetro do método drawString deve obrigatoriamente ser do tipo *string*. Assim, é feita uma conversão de tipos através do método Integer.toString(int numero), que retorna o número inteiro convertido para uma cadeia de caracteres. Para se fazer o processo inverso, ou seja, converter um string para um número inteiro, deve-se usar o comando Integer.parseInt (String palavra), porém estando atentos para que a palavra seja um número no formato string, já que não faria sentido converter uma palavra qualquer para número.

8.3. O Game Loop

Embora existam muitos supermercados diferentes em nossa cidade, ao entrar num deles imediatamente sabemos que se trata de um estabelecimento

desse tipo, já que todos têm uma série de características comuns, tais como várias prateleiras com produtos, caixas atendentes, carrinhos de supermercado etc. Em computação ocorre algo parecido: há diversos tipos de sistemas, mas eles podem ser agrupados em conjuntos de sistemas que seguem uma arquitetura semelhante entre si: sistemas de banco de dados possuirão sempre um esquema parecido, por mais diferente que seja o propósito entre eles. Em jogos isso também ocorre: desde um simples *Pong* até um jogo 3D de última geração, eles possuem uma arquitetura semelhante: todos seguem a arquitetura de um *game loop*, sendo cada ciclo desse *loop* responsável por produzir uma imagem do jogo, relativo a um quadro.

A grande maioria dos motores de jogos implementa essa arquitetura básica. No javaPlay o game loop pode ser resumido através do diagrama da Figura 8.4:

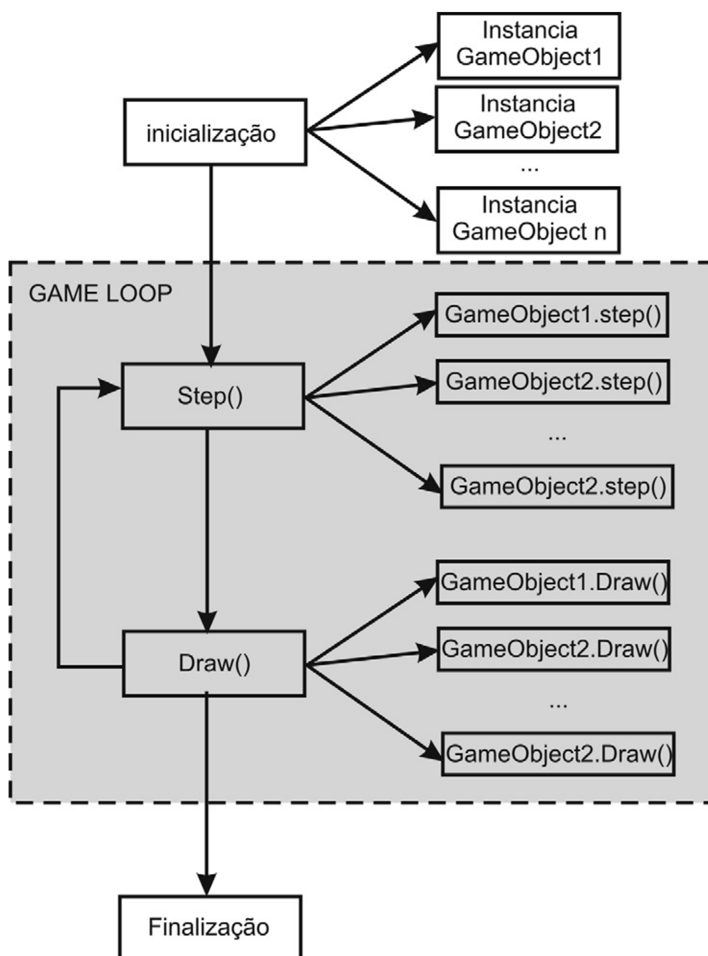


Figura 8.4 Arquitetura do javaPlay.

Embora, na maioria das vezes, o desenvolvedor de jogos não precise implementar o *game loop*, uma vez que o motor estará fazendo isso para ele, ao menos uma vez na vida é muito saudável que a tarefa seja feita. Nesta seção desenvolveremos um *game loop* básico, em que substituiremos o *game object* por um simples objeto gráfico, e o exemplo consistirá em fazer uma bolinha mover-se pela tela. O *step* corresponderá a atualizar um valor de posição desse objeto na tela, e o passo de *draw* corresponderá a fazer a chamada ao método do objeto *graphics* propriamente dito, responsável por desenhar o elemento em questão.

Para tanto, a primeira tarefa é iniciar o modo gráfico do programa. Como o procedimento é sempre o mesmo, em situações normais deixaremos que o javaPlay cuide desse processo. Entretanto, vamos neste exemplo evidenciar a inicialização.

Primeiramente devemos criar um GraphicsEnvironment, que é uma classe abstrata. Este será responsável pelos conhecimentos dos dispositivos de saída do sistema. Através de um método dessa classe, chamado getDefaultScreenDevice, será feito um mapeamento da aplicação para o dispositivo de tela a ser usado pela aplicação. Finalmente, a classe GraphicsConfiguration armazenará toda a configuração gráfica do dispositivo. Não é necessário que você se preocupe em entender a fundo como são cada uma das classes, uma vez que elas irão implementar ações e recursos que são de baixo nível para a aplicação em si. Em parte, serão elas que irão garantir que a aplicação gráfica funcione independentemente da plataforma, placa gráfica ou tipo de dispositivo de saída.

Uma vez feito esse processo de inicialização gráfica, será necessário criar a tela de desenho, também chamada de *Canvas*, que será como o papel no qual estaremos desenhando todos os objetos gráficos. Sem um *canvas,* o objeto *graphics* não saberá como desenhar, pois é como dar-lhe uma ordem de pintura sem estabelecer o local para tal. Embora o Java ofereça uma classe chamada *Canvas*, usaremos neste exemplo uma classe muito semelhante fornecida pelo javaPlay, chamada de GameCanvas. A linha 22 do Código 8.11 mostra esse passo. Ao criar o *canvas*, é importante dizer qual será a resolução dessa tela. Neste exemplo deixaremos que o javaPlay coloque o valor padrão, que é 800 x 600.

Código 8.11 Implementação do Game Loop

```
package demo;

import javaPlay.GameEngine;
import javaPlay.GameCanvas;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.GraphicsConfiguration;
import java.awt.GraphicsDevice;
import java.awt.GraphicsEnvironment;

public class Main
{
   public static void main(String[] args)
   {
      GraphicsEnvironment graphEnv =
         GraphicsEnvironment.getLocalGraphicsEnvironment();
      GraphicsDevice graphDevice =
         graphEnv.getDefaultScreenDevice();
      GraphicsConfiguration graphicConf =
         graphDevice.getDefaultConfiguration();

      GameCanvas tela = new GameCanvas(graphicConf);

      int x = 50;  // Esta parte corresponde a etapa de inicialização
      int y = 50;
      int velX = 2;
      int velY = 1;

      while(true)     // Este é o Game Loop…
      {
         Graphics g = tela.getGameGraphics();
         g.setColor(Color.black);
         g.fillRect(0, 0, 800,600);
         g.setColor(Color.green);

         x += velX;
         y += velY;

         if (((x+20) > 800) || (x < 0)) velX *=-1;
         if (((y+20) > 600) || (y < 0)) velY *=-1;

         g.fillOval(x, y, 20, 20);  // 20 x 20 é o tamanho da bola
            tela.swapBuffers();
      }
   }
}
```

Há um detalhe muito importante a ser esclarecido em relação ao *canvas*: o dispositivo gráfico possui uma área na memória chamada de *front buffer*. Tudo o que enviamos ao *front buffer* aparece imediatamente na tela. Ao longo de uma aplicação gráfica desenharemos diversos objetos. A operação de desenho será feita através de uma sequência linear de operações: pedimos para desenhar um elemento depois do outro – muito embora, quando se está trabalhando com 3D, possam ser enviados muitos elementos gráficos de uma vez

só para serem pintados na tela pela placa gráfica. Se a aplicação fosse desenhar o resultado diretamente no *front buffer* ocorreria um efeito indesejado: objetos piscando na tela. Isso ocorre porque num mesmo frame objetos seriam desenhados aos poucos na tela, e tão logo o último fosse enviado para o *frame buffer*, este seria limpo e novamente começariam a surgir aos poucos outras figuras. Para resolver o problema, existe uma memória de vídeo secundária, chamada de *back buffer*. Todos os pedidos de desenho serão feitos nessa área de memória. Entretanto, ela não é visível, portanto tudo o que desenhamos no *back buffer* não aparecerá de imediato na tela. Ao terminar de desenhar todos os elementos de um determinado frame, será realizada uma troca (*swap*) de *buffers*, copiando o conteúdo do *back buffer* para o *front buffer* e liberando o espaço do primeiro para que uma nova imagem comece a ser pintada. Enquanto a imagem é pintada novamente no *back buffer*, a aplicação fica mostrando a imagem do *front buffer*, que agora não será alterada e, portanto, durante algumas frações de segundo, tempo correspondente para que um novo frame seja produzido no *back buffer*, ficará estática.

Uma vez criado o *canvas*, podemos começar a preocuparmo-nos com a aplicação em si. Antes de entrar no *game loop*, seguindo a arquitetura ilustrada na Figura 8.4, realizaremos a etapa de inicialização. Para este exemplo, inicializaremos os atributos da bolinha correspondente a sua posição e a sua velocidade, uma para cada eixo, conforme pode ser visto no Código 8.11, linhas 24 a 27. Finalmente inicia-se o *game loop*.

O *game loop* consiste em criar uma referência ao objeto *Graphics* (linha 31), configurar a cor de pintura de fundo (linha 32), pintar a tela de fundo (linha 33), configurar a cor do objeto gráfico (linha 34), atualizar a posição da bolinha, conforme a sua velocidade em cada eixo (linhas 36 e 37), verificar se ela se chocou com as paredes, e em caso positivo inverter o sentido do seu movimento, invertendo o sentido da velocidade (linhas 39 e 40), desenhar a bolinha (linha 42) e finalmente realizar a operação de *Swap Buffer*, conforme explicado anteriormente (linha 43).

8.4. Incrementando o modo gráfico

Com o que foi apresentado até o momento é impossível desenvolver um jogo que tenha uma arte mais sofisticada. Isso ocorre porque apenas

desenhar retângulos, círculos ou textos é extremamente limitado para uma produção artística mais elaborada. Assim, para um jogo de verdade, é fundamental que se usem recursos externos, também chamados de *Assets*, que nada mais são que objetos produzidos externamente e trazidos ao ambiente de desenvolvimento. São típicos exemplos de *assets*: modelos 3D, sons, imagens e texturas. Dentro do universo de jogos 2D, os *assets* mais usados serão imagens, que servirão para representar *sprites*, plano de fundo, objetos parados, dentre outros elementos. Os *sprites* são elementos comuns em jogos 2D e podem ser definidos como imagens que ocupam uma parte da área gráfica, com certa independência uns dos outros, como foi mencionado no início desta seção.

Os *sprites* não precisam ter um contorno retangular, como costuma acontecer com uma imagem. Para tanto, alguns formatos de imagens digitais permitem que para cada pixel, além do seu RGB, haja também um elemento chamado de *Alpha*. Esse quarto valor para um pixel indica o quanto ele é transparente, fazendo com que partes da imagem não sejam visíveis. Não são todos os formatos de imagem que permitem guardar tal informação. Exemplos de formatos que suportam transparência são o PNG, TGA e TIFF. É bom ter em conta que o famoso JPG não permite esse recurso e é por isso que não é tão usado pelos artistas de jogos. No javaPlay, sempre que uma imagem com *alpha* for usada, a sua transparência será ativada. Imagens que carregam informação de transparência normalmente são chamadas de imagens de 32 bits, já que serão necessários 4 bytes para representar a informação de cada pixel, ao contrário dos 3 bytes (ou 24 bits) quando apenas se possui o RGB.

Outra característica dos *sprites* é o fato de poderem ter animação. Inicialmente essa animação poderia consistir em termos uma grande coleção de imagens que seriam trocadas rapidamente ao longo da aplicação. Entretanto, tal abordagem nos traria uma tarefa relativamente custosa para armazenar um número grande de imagens, tendo de provavelmente usar um vetor de imagens para cada *sprite*. Outra abordagem, que será usada pelo javaPlay, consiste em criar "tiras de animação". Numa mesma imagem colocaremos toda a sequência de um *sprite* animado, tal como pode ser visto na Figura 8.5.



Figura 8.5 Tiras de animação para um *sprite.*

É importante notar que, ao mostrar um *sprite* na tela, será necessário mostrar apenas um pedaço da imagem, correspondente ao quadro da animação em questão. Assim, supondo que a largura da imagem seja de 472 pixels e a imagem possui 8 quadros de animação, como na Figura 8.5, cada um dos *sprites* possuirá 59 pixels. Assumindo que os índices dos *sprites* vão de 0 a *n*-1, sendo *n* o número total de quadros da sua animação, teremos a seguinte equação para dizer qual o início e o fim da coordenada de largura do *sprite i* a ser mostrado:

Canto esquerdo do *sprite i* = *i* * larguraDoSprite

E a coordenada final do *sprite* a ser mostrado será dada por:

Canto direito do *sprite i* = *i* * larguraDoSprite + larguraDoSprite.

Embora possa parecer um pouco complicado, veremos que no javaPlay essas contas estarão encapsuladas pela classe que lidará com os *sprites*.

Antes de entender como lidar com *sprites*, é importante mencionar que também é a classe *Graphics* que será responsável por lidar com imagens. Isso significa que ela é capaz de carregar e mostrar imagens na tela da aplicação, tarefa que pode ser feita pelo seguinte método:

g.drawImage (Image figura, int posicaoX, int posicaoY);

Tal como ocorreu em relação às cores e às fontes, para lidar com imagens será usada uma outra classe juntamente com a *Graphics*. Trata-se da classe *Image*, que basicamente possuirá um arquivo de imagem na sua estrutura.

Porém, em vez de vermos exemplos mais elaborados do uso da classe *Image* e a própria classe *Graphics* para desenhar imagens, a partir deste momento estaremos utilizando algumas facilidades que o motor javaPlay irá trazer.

Para lidar com as imagens de uma forma mais intuitiva, o javaPlay implementa uma classe chamada *Sprite.* Essa classe é muito parecida com uma imagem, mas implementará algumas facilidades no que se refere a lidar com animação. Vejamos a seguir os atributos da classe *Sprite*:

public int x, y
Coordenada x, y do canto superior esquerdo do *sprite* na tela (herdada da imagem).

public Image image
A classe imagem que representará o *sprite*

public int width, height
Largura e Altura do *Sprite*

E os métodos:

public Sprite(String filename, int animFrameCount, int animFrameWidth, int animFrameHeight)
Construtor de um *sprite*. Recebe o nome do arquivo e, no caso de ser um *sprite* animado, um contador animFrameCount dizendo quantos frames o *sprite* possui, a largura animFrameWidth e a altura animFrameHeight de cada frame do *sprite*.

public void setCurrAnimFrame(int frame)
Indica qual é o frame do *Sprite* animado a ser usado num determinado instante.

public void draw(Graphics g, int x, int y)
Desenha o *sprite* no *canvas*, com o seu canto superior esquerdo na posição x, y da tela.

Como já foi mencionado, a versão completa do motor javaPlay define os elementos de um jogo como sendo *game objects*. Em jogos 2D, a grande maioria dos *game objects* poderá ser visível na tela na forma de um *sprite*. Assim, ao definir um *game object* será usual criar um atributo *sprite* para ele, além dos atributos típicos para um *game object* em questão.

Neste capítulo estenderemos o conceito do javaPlay, porém usando agora a sua real arquitetura, que consiste em implementar cada objeto de jogo separadamente, através de uma extensão do *game object*. Essa estratégia é muito importante na engenharia de um jogo, pois possibilitará que ele possa ser modularizado corretamente e de forma elegante.

De forma geral, a arquitetura do javaPlay pode ser representada pelo diagrama da Figura 8.6:

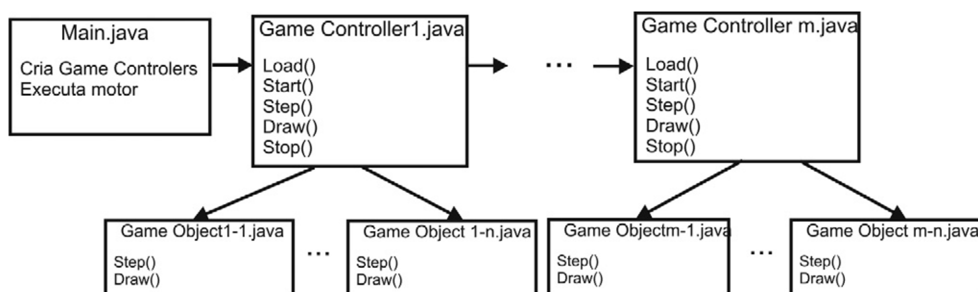


Figura 8.6 Diagrama resumido da arquitetura do javaPlay.

O módulo principal do javaPlay define uma série de objetos da classe *Game Controller*. Para facilitar a compreensão, podemos imaginar que cada *Game Controller* corresponderá a uma fase ou cena de um jogo. Esse tipo de objeto pode ser usado também para elementos de interfaces e telas intermediárias. Cada *Game Controller* terá seu próprio método de inicialização, que será implementado dentro do Load(). É nesse método que serão instanciados todos os *game objects* de uma fase. O método Start() será chamado pelo motor uma única vez, após a execução do método load(). Dentro desse método serão inicializados todos os atributos globais que serão usados ao longo da fase. Tal como os *game objects*, os *Game Controllers* possuem um método de step() e draw(), chamados uma vez a cada passo do *game loop*. É dentro do step() que implementaremos a lógica de cada fase do jogo. Caso alguns *game objects* contenham a lógica dentro do seu próprio método step, é aqui que deverão ser chamados. A diferença do step do *Game Controller* para o step do *Object Controller* consiste no fato de que dentro do primeiro é possível ver os atributos e métodos de cada um dos *game objects.* Já no step de um *game object* apenas é possível ver os atributos dele mesmo.

Através do exemplo que será mostrado a seguir, você poderá entender melhor a arquitetura, que num primeiro momento pode parecer um pouco complexa. Implementaremos uma parte do Pong, que chamaremos de "pré-Pong". Para tanto, estaremos usando *sprites*, *game objects* e um *Game Controller*. O jogo ainda não estará completo, pois ainda faltarão os controles do jogador, que serão vistos no capítulo seguinte. Em vez de permitir que um jogador controle cada barra, o programa as moverá numa velocidade constante. A Figura 8.7 ilustra o protótipo.



Figura 8.7 Esquema do protótipo do PONG a ser construído.

Neste protótipo serão criados três *game objects*, um para a bolinha e outros dois para cada barra. Neste protótipo haverá apenas um *Game Controller*, denominado de SIMPLE_CONTROLLER. O Código 8.12 mostra o módulo Main.Java.

Código 8.12 Modulo Main.Java do Pré-Pong

```
1   import javaPlay.GameEngine;
2   public class Main
3   {
4      public static void main(String[] args)
5      {
6         GameEngine.getInstance().addGameStateController
7            (Global.SIMPLE_CONTROLLER_ID, new SimpleController());
8         GameEngine.getInstance().setStartingGameStateController
9            (Global.SIMPLE_CONTROLLER_ID);
10        GameEngine.getInstance().run();
11     }
12  }
```

A responsabilidade do módulo Main está basicamente em instanciar os *Game Controllers* (linha 6), especificar qual é o primeiro *Game Controller* do jogo, que será chamado logo no início (linha 8) e finalmente executar o motor (linha 10).

Antes de ver o *SimpleController* em detalhes, veremos como é cada um dos *game objects*. O Código 8.13 implementa o *game object* da bolinha. Como se pode observar, a classe bolinha será uma extensão de um *game object* padrão. Ao fazer isso, estaremos herdando os atributos de posição *x* e *y*, bem como os métodos step() e draw(), que implementam a lógica e o desenho específico de cada *game object*. Para a bolinha, criamos três atributos: um *sprite*, sua velocidade no eixo x e sua velocidade no eixo y. Neste exemplo estaremos optando por implementar a lógica da bolinha no step do *Game Controller*, pois precisaremos de informações de outros *game object*, especialmente para tratar a colisão da bolinha com as barras, portanto o método step está vazio. Já o método de desenhar a bolinha se resume a desenhar o *sprite* em questão, dessa forma, apenas chamamos o método de desenho do *sprite* que está sendo usado para a bolinha.

Código 8.13 Game Object bolinha

```
import java.awt.Graphics;
import javaPlay.GameObject;
import javaPlay.GameEngine;
import javaPlay.Sprite;

public class bolinha extends GameObject
{
   public Sprite sprite;
   public int velX = 1;
   public int velY = 1;

   public void step(long timeElapsed)
   {
   }

   public void draw(Graphics g)
   {
      sprite.draw(g, x, y);
   }
}
```

Embora haja duas barras, ambas terão as mesmas características, assim, apenas uma classe será necessária para esses *game objects*. O Código 8.14 implementa a classe.

Código 8.14 Game Object da barra

```
import java.awt.Graphics;
import javaPlay.GameObject;
import javaPlay.Sprite;

public class barra  extends GameObject{

   public Sprite sprite;
```

Código 8.14 Game Object da barra (cont.)

```
    public int velY = 3;

    public void step(long timeElapsed)
    {
    }

    public void draw(Graphics g)
    {
      sprite.draw(g, x, y);
    }
}
```

Os únicos parâmetros dessa extensão de um *game object* são o *sprite* que será usado e a velocidade de movimento na vertical. O método de step será implementado no *Game Controller,* e o método de desenho basicamente consiste em solicitar que o *sprite* seja desenhado na posição *x, y* correspondente.

Neste exemplo, toda a lógica da aplicação está implementada no step do *Game Controller*. O código completo desse *Game Controller* não será listado no livro, para economizar espaço, porém pode ser encontrado por completo no material suplementar. Entretanto, a seguir serão listados e detalhados alguns dos seus métodos. O Código 8.15 detalha o método load. Inicialmente são carregados os três *sprites*, um para cada *game object* do Pong. Como estamos usando *sprites* em vez de imagem, temos de dizer o número de imagens animadas que o *sprite* possui, bem como o tamanho de cada parte da imagem. Como essas imagens não possuem animação, estamos dizendo que apenas há um *frame* e o tamanho é exatamente o mesmo da imagem em si. Em seguida, criamos um objeto chamado fundo, que é uma instância da classe *Background*. Essa classe é muito semelhante à classe *Sprite*, porém não lida com animação. Na linha 9 cria-se uma instância da bolinha e logo em seguida associa-se o *sprite* criado anteriormente para o atributo correspondente. O mesmo processo será feito para as barras da esquerda e da direita. Finalmente, será armazenada a resolução do *Game Canvas* que está sendo criado. Perceba-se que esse valor irá permanecer constante ao longo de toda a aplicação, uma vez que a leitura está sendo feita no método load, executado apenas uma vez. Assim, caso o jogador resolva alterar o tamanho da janela durante a execução da aplicação, esses atributos não serão mais alterados. Para nos prevenir contra esse possível "ataque" do usuário, teríamos de fazer a leitura de resolução na função step(), fazendo com que a cada novo frame seja feita uma nova leitura da resolução. Por motivos óbvios, essa segunda abordagem pode prejudicar um pouco a performance do jogo...

Código 8.15 Método load do Game Controller do pré-Pong

```
public void load()
{
   figuraBola = new Sprite(“ball.png”, 1, 34, 34);
   figuraBarra1 = new Sprite(“pad01.png”, 1, 25, 100);
   figuraBarra2 = new Sprite(“pad02.png”, 1, 25, 100);

   fundo = new background(“background.png”);

   bola = new bolinha();
   bola.sprite = figuraBola;

   barraEsquerda = new barra();
   barraEsquerda.sprite= figuraBarra1;

   barraDireita = new barra();
   barraDireita.sprite=figuraBarra2;

   alturaTela = GameEngine.getInstance().getGameCanvas().
      getHeight();
   larguraTela = GameEngine.getInstance().getGameCanvas().
      getWidth();
}
```

O método start do *Game Controler* é simples e será responsável por inicializar atributos gerais. Neste exemplo, apenas inicializamos as posições iniciais da bolinha e das barras. O método step do *Game Controller*, listado no Código 8.16, será o responsável pela lógica da aplicação em si.

Código 8.16 Método step do *Game Controller* do pré-Pong

```
public void step(long timeElapsed)
   {
      // movimento da bolinha
      bola.x += bola.velX;
      bola.y += bola.velY;

      if ((bola.x + bola.sprite.width > larguraTela)||
          (bola.x<0)) bola.velX *= -1;
      if ((bola.y + bola.sprite.height > alturaTela)||
          (bola.y<0)) bola.velY *= -1;

      //movimento da barra esquerda
      barraEsquerda.y += barraEsquerda.velY;
      if (( barraEsquerda.y + barraEsquerda.sprite.height >
            alturaTela)||( barraEsquerda.y < 0))
      barraEsquerda.velY *= -1;

      // movimento da barra direita
      barraDireita.y += barraDireita.velY;
      if (( barraDireita.y + barraDireita.sprite.height >
            alturaTela)||(barraDireita.y < 0))
      barraDireita.velY *= -1;
}
```

A cada frame a posição da bola será incrementada de acordo com a sua velocidade. Tão logo essa posição seja atualizada, será verificado se o canto direito da bolinha (bola.x + bola.sprite.width) ultrapassou o limite horizontal direito da tela (larguraTela) ou se o canto esquerdo da bolinha (bola.x) ultrapassou o limite horizontal esquerdo da tela (coordenada 0). Se isso ocorrer, a velocidade da bolinha será invertida, de forma que no frame seguinte a bolinha começará um movimento no sentido contrário. O mesmo é feito em relação às coordenadas de altura, verificando se a bolinha ultrapassou o limite superior e inferior da tela. Esses procedimentos estão listados nas linhas 7 a 10. Como neste exemplo as barras estarão se movendo sozinhas (futuramente será inserida uma entrada de dados do teclado ou mouse), aplica-se a mesma lógica da bolinha para cada uma das barras. No entanto, neste caso apenas movemos a barra na vertical. O método draw do *Game Controller* é relativamente simples e apenas invocará o método draw de cada um dos *game object*, bem como o *draw* do objeto fundo. É importante notar que a ordem com que se invoca esses métodos será importante. Como queremos que todos os *game objects* se sobreponham ao fundo, chamaremos o método de desenho do fundo em primeiro lugar. Caso o chamássemos por último, ele se desenharia por cima dos demais *sprites*.

8.5. Colisão

A bolinha e as barras estão se movendo livremente, com total independência entre si. No jogo de verdade precisamos tratar da colisão entre a bolinha e as barras. Tratamento de colisão é uma das tarefas que tipicamente são chamadas de física em desenvolvimento de jogos. Tratar choques em objetos 3D não é uma tarefa computacional muito trivial e, se não for feito com cuidado, pode consumir um grande tempo de processamento do computador. Felizmente, o tratamento de colisões não é uma tarefa muito complexa quando se está no mundo 2D...

No universo material um choque entre objetos se dá pelo fato de que dois corpos diferentes não podem ocupar o mesmo espaço ao mesmo tempo. No mundo virtual isso é possível de acontecer, já que não existe matéria... A ideia do algoritmo de colisão consistirá em efetuar um teste que simule a lei física mencionada. Assumiremos que um choque consiste em que um pedaço de um corpo esteja começando a entrar no outro corpo. O tratamento de cho-

que consistirá em evitar que esse movimento seja concluído, evitando, assim, a penetração.

Como o universo que estamos criando é basicamente constituído de *sprites*, portanto imagens, teremos mais um fator que simplificará o cálculo do choque: todos os objetos são retângulos. Assumindo essa premissa podemos afirmar que "se houver a colisão entre dois *sprites*, pelo menos um dos vértices de um dos *sprites* estará dentro da área do outro *sprite*". Caso não tenha entendido a frase, procure fazer o seguinte exercício: procure desenhar dois *sprites* que estão em processo de colisão, um penetrando no outro, mas que nenhum dos vértices de um *sprite* esteja dentro do outro *sprite*. Mas não fique tentando por muito tempo, pois como poderá perceber, é uma situação impossível... A Figura 8.8 ilustra o processo de colisão entre o *sprite* 1 e o *sprite* 2. Perceba que tanto o vértice do canto inferior direito do *sprite* 1 como o vértice do canto superior esquerdo do *sprite* 2 estão "invadindo" a área do outro.

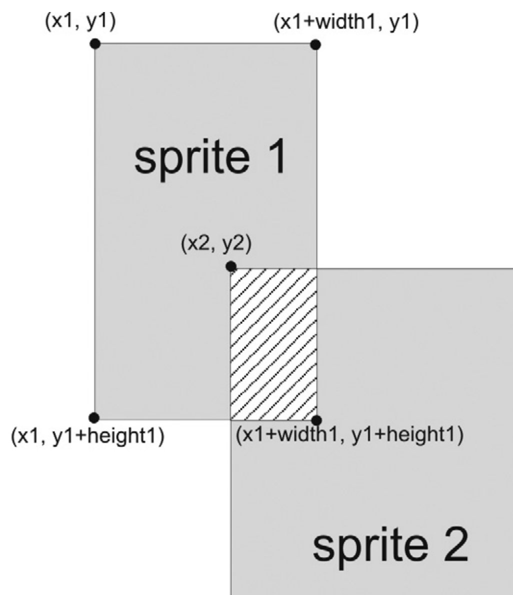


Figura 8.8 Colisão entre dois *sprites*.

Sendo o parâmetro width1 e height1 o comprimento e a altura de um *sprite*, seus quatro vértices podem ser descritos pelas seguintes coordenadas: (x1, y1), (x1+width1, y1), (x1, y1+height1) e (x1+width1, y1+height1). Verificar se um *sprite* está dentro deste consiste em verificar se um dos quatro vértices do segundo *sprite* recai dentro do retângulo definido pelos quatro vértices

citados. A expressão a seguir retornará verdade, caso o vértice (x2, y2) esteja dentro da área do primeiro retângulo:

(x1 < x2 < x1+width1) e (y1 < y2 < y1+height1)

Entretanto, apenas esse teste não é suficiente para concluir que o primeiro *sprite* está colidindo com o outro. Isso ocorre porque pode ser que outro vértice do segundo retângulo esteja penetrando a área do primeiro retângulo, tal como pode ser visto na Figura 8.9, (a). Neste exemplo, o vértice superior esquerdo não está entrando no *sprite* 1, porém os vértices do lado direito estão, portanto, há colisão entre ambos. Para resolver a situação um teste de colisão mais correto consiste em verificar se pelo menos um dos quatro vértices do segundo retângulo está dentro do primeiro. Assim, a expressão se estenderia para:

(x1 < x2 < x1+width1) E (y1 < y2 < y1+height1)
Ou (x1 < x2 + width2 < x1+width1) E (y1 < y2 < y1+height1)
Ou (x1 < x2 < x1+width1) E (y1 < y2 +height2 < y1+height1)
Ou (x1 < x2 + width2 < x1+width1) E (y1 < y2 +height2 < y1+height1)

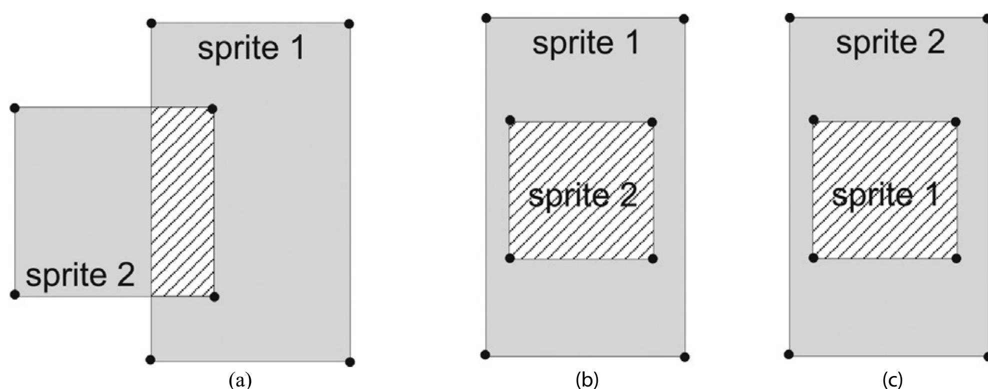


Figura 8.9 Diferentes configurações para colisões entre sprites.

Por fim, há uma situação em que não seria possível resolver essa sequência de expressões, como mostra a Figura 8.9 (c): o *sprite* 1 está dentro do *sprite* 2. Nesse caso, perceba-se que todos os vértices do *sprite* 2 estarão fora do *sprite* 1, no entanto, há uma situação clara de colisão. Assim, o teste completo de colisão também deve verificar se os vértices do *sprite* 1 estão dentro do *sprite* 2, tal como pode ser visto na expressão final de colisão:

(x1 < x2 < x1+width1) E (y1 < y2 < y1+height1)
Ou (x1 < x2 + width2 < x1+width1) E (y1 < y2 < y1+height1)

Ou $(x1 < x2 < x1+width1)$ E $(y1 < y2 +height2 < y1+height1)$
Ou $(x1 < x2 + width2 < x1+width1)$ E $(y1 < y2 +height2 < y1+height1)$
Ou $(x2 < x1 < x2+width2)$ E $(y2 < y1 < y2+height2)$
Ou $(x2 < x1 + width1 < x2+width2)$ E $(y2 < y1 < y2+height2)$
Ou $(x2 < x1 < x2+width2)$ E $(y2 < y1 +height1 < y2+height2)$
Ou $(x2 < x1 + width1 < x2+width2)$ E $(y2 < y1 +height1 < y2+height2)$

O Código 8.17 traduz essa expressão para o Java, através de uma função que recebe as coordenadas do canto superior esquerdo de cada retângulo, bem como seu comprimento e largura de dois elementos, retornando *true* caso ocorra colisão e *false* caso contrário.

Código 8.17 Função para o cálculo de colisão

```
public boolean collision(int x1, int y1, int width1, int height1,
                         int x2, int y2, int width2, int height2)
{
   if (((x1 < x2) && (x2 < (x1+width1))
         && ((y1 < y2)&&(y2<(y1+height1))))
      || ((x1 < x2+width2) && (x2+width2 < (x1+width1))
         && ((y1 < y2)&&(y2<(y1+height1))))
      || ((x1 < x2) && (x2 < (x1+width1))
         && ((y1 < y2+height2)&&(y2+height2<(y1+height1))))
      || ((x1 < x2+width2) && (x2+width2 < (x1+width1))
         && ((y1 < y2+height2)&&(y2+height2<(y1+height1))))
      || ((x2 < x1) && (x1 < (x2+width2))
         && ((y2 < y1)&&(y1<(y2+height2))))
      || ((x2 < x1+width1) && (x1+width1 < (x2+width2))
         && ((y2 < y1)&&(y1<(y2+height2))))
      || ((x2 < x1) && (x1 < (x2+width2))
         && ((y2 < y1+height1)&&(y1+height1<(y2+height2))))
      || ((x2 < x1+width1) && (x1+width1 < (x2+width2))
         && ((y2 < y1+height1)&&(y1+height1<(y2+height2)))))

      return (true);
   else return (false);
   }
}
```

8.6. Opsss... tratamento de exceção

Bugs e erros no programa são uma constante na vida do programador. Entretanto, o usuário final não quer e não deve ter de se preocupar com esse tipo de problemas. Se o usuário estiver jogando e no meio da partida aparecer uma janela dizendo "Vou parar o programa, pois houve uma tentativa de divisão por zero", provavelmente o jogador irá querer o dinheiro gasto com o software de volta...

Assim, a arte de programar bem também requer saber tratar erros de uma forma elegante. Enquanto os erros de sintaxe são facilmente detectáveis, ao ponto de que o programa nem sequer é capaz de compilar caso não seja corrigido, outros erros podem ocorrer durante a execução do programa. Suponha que o jogo pergunte qual é o nível de dificuldade que o jogador quer: 1, 2 ou 3. No entanto, o usuário rebelde digita a letra "d". O que o programa deverá fazer? O dígito não é nenhuma das opções indicadas e nem sequer é do tipo numérico, como o esperado.

O Java possui um mecanismo muito poderoso, chamado tratamento de exceções, para que o programador seja capaz de lidar com erros que podem ocorrer durante a execução de um programa. Chamamos de exceção a algo que é um pouco mais genérico do que um erro. Em algumas situações uma exceção pode efetivamente ser um erro, mas em outras pode acontecer de ser apenas um caso que se deve precaver.

Sem um tratamento de exceção, se um erro fosse detectado o programa simplesmente abortaria, mostrando alguma mensagem feia e provavelmente chateando o usuário. O Java permite que esse erro seja colhido (*catch*) e observado com mais cuidado, para ver o que se pode fazer, evitando que o programa seja abortado ou que o usuário seja insultado. Esse tema é bem mais complexo do que será exposto neste capítulo, sendo conveniente que o usuário mais experiente em Java procure uma documentação mais detalhada.

As exceções em Java serão tratadas a partir de um objeto do tipo *Exception*. Essa classe possui várias subclasses, que são casos mais particulares de exceção. A título de exemplo, a subclasse *NumberFormatException* é uma exceção de um formato não-válido. Assim, se chamarmos a função Integer.parseInt("erro") criaremos uma exceção desse tipo, uma vez que o formato esperado pela função em questão é uma string que possa ser convertida em número, mas a palavra "erro" nunca poderá se tornar um número.

Quando uma exceção ocorre, o Java irá jogar (*throw*) uma exceção, isto é, criar um objeto do tipo do erro que ocorreu. Quando uma exceção é jogada, pode-se colhê-la (*catch*), de forma a fazer algum tratamento adequado. Isso é feito mediante o uso dos comandos *try-catch*, seguindo a sintaxe do Código 8.18.

Quando o Java encontrar um escopo de *try,* ele irá se precaver e saber de antemão que será executada uma expressão que tem a possibilidade de conter uma exceção. Se, após executar o trecho de programa que está dentro

do *try,* o computador não encontrar nenhum erro, o fluxo do programa continuará normalmente, pulando-se o trecho de *catch*. Entretanto, se dentro desse espaço ocorrer um erro que seja de algum dos tipos descritos pelas classes ClasseDeExceção1, ..., ClasseDeExceçãoN, então o fluxo do programa será desviado para o escopo do *catch* correspondente a esse tipo de erro, executando, assim, a expressão correspondente.

Código 8.18 Sintaxe do tratamento de exceção com try-catch

```
try {
   expressão
}
catch ( ClasseDeExceção1  nomeDaVariavelDeExceção1 ) {
   expressãoErro1
catch ( ClasseDeExceção2  nomeDaVariavelDeExceção2 ) {
   expressãoErro2
...
catch ( ClasseDeExceçãoN  nomeDaVariavelDeExceçãoN ) {
   expressãoErroN
}
```

Ao entrar num dos blocos de exceção, podemos criar uma variável de exceção que eventualmente poderá ser usada ao longo do tratamento. O Código 8.19 ilustra como seria o tratamento de exceção para o caso de um valor a ser convertido em número não ser válido para tal.

A fim de que o Java seja capaz de "adivinhar" qual é o tipo de exceção que ocorreu é necessário fazer com que uma função específica seja capaz de detectar que houve um erro, e, em caso positivo, essa função deve jogar (*throw*) uma instância do objeto de exceção. Embora não precisemos remeternos ao código-fonte da função Integer.parseInt(), devemos saber de antemão que o programador de tal função realizou um teste para verificar o conteúdo do parâmetro; caso este não seja dígitos numéricos, foi feito um *throw* de uma classe de exceção adequada. Vejamos outro exemplo simples. Suponha que devemos ler dois números do usuário e realizar uma divisão entre ambos, mostrando logo em seguida o resultado ao usuário. Entretanto, a matemática nos diz que não é possível dividir qualquer número por zero. Assim, a função de divisão realiza o lançamento (*throw*) de um objeto de exceção caso isso ocorra, conforme pode ser visto nos trechos do Código 8.20, linha 4: um objeto do tipo Exception é criado e devolvido como resultado dessa função. Quando esta for chamada por algum trecho do programa (linha 12), caso ocorra o erro previsto, a função retornará um objeto de exceção que será levado ao seu *catch* correspondente. Observe que esse mecanismo de diferentes objetos do

tipo *Exception* permite que sejam tratados diferentes casos de erro, tal como no exemplo do Código 8.20.

Código 8.19 Exemplo de tratamento de exceção para conversão de string para número

```
String leitura;
leitura = TextIO.getlnWord();
try {
   double x;
   x = Double.parseDouble(leitura);
   System.out.println( "o número é " + x );
}
catch ( NumberFormatException e ) {
   System.out.println( "o valor entrado não parece ser um
      número..." );
}

```

Código 8.20 Tratamento de exceção por parte de uma função

```
public double divisao (int numerador, int denominador) throws
   Exception
{
   if (denominador == 0)
      throw new Exception();
   return (double) (numerador / denominador);
}
...
try
{
   int numero1 = Integer.parseInt(TextIO.getlnWord());
   int numero2 = Integer.parseInt(TextIO.getlnWord());
   int result = divisao (numero1, numero2);
}
catch (NumberFormatException format)
{
   System.out.print("erro de formato");
}
catch (Exception porZero)
{
   System.out.print("Tentativa de divider por zero");
}
System.out.print("resultado da divisão:"+result);
...
```

Código 8.21 Método load do *Game Controller* do Pré-Pong.

```
public void load()
{
   try
   {
      figuraBola = new Sprite("ball.png", 1, 34, 34);
      figuraBarra1 = new Sprite("pad01.png", 1, 25, 100);
      figuraBarra2 = new Sprite("pad02.png", 1, 25, 100);
      fundo = new background("background.png");
   }
   catch (Exception ex)
   {
      System.out.print("erro de leitura de imagem");
```

Código 8.21 Método load do *Game Controller* do Pré-Pong (cont.)

```
            }
            bola = new bolinha();
            bola.sprite = figuraBola;
            barraEsquerda = new barra();
            barraEsquerda.sprite= figuraBarra1;
            barraDireita = new barra();
            barraDireita.sprite=figuraBarra2;
            alturaTela = GameEngine.getInstance().getGameCanvas().
               getHeight();
            larguraTela = GameEngine.getInstance().getGameCanvas()
               getWidth();
        }
```

O javaPlay possui um tratamento de exceção para a criação de um *sprite*. Caso a imagem que será usada tenha algum tipo de problema, será gerada uma exceção que poderá eventualmente ser tratada pelo programador. O Código 8.21 mostra como o Código 8.15 pode ter incorporado em si esse tratamento de exceção. Neste exemplo, apenas se imprime uma mensagem, porém outro tipo de tarefa mais complexa poderia ser feito, tal como carregar uma imagem default, que não comprometesse o funcionamento da aplicação em si.

EXERCÍCIOS

E8.1 Incremente o método step do Código 8.9 para que a bolinha aumente de velocidade cada vez que rebater na parede da esquerda. Tome a devida precaução para que, a partir de um determinado valor da velocidade ela não continue crescendo, de forma a evitar que seu movimento seja extremamente rápido e praticamente impossível de ser visto.

E8.2 Acrescente ao Código 8.11 duas barrinhas no estilo Pong que se movam aleatoriamente.

E8.3 Consultando a documentação do Java, utilize o Código 8.11 para desenhar uma imagem em vez do círculo, usando diretamente as classes *Graphics* e *Image*.

E8.4 Quando dois objetos que estão em movimentos separados colidem, pode ocorrer a transmissão de parte do movimento de um para outro. Em futebol chamamos este fenômeno de “efeito”. Usando a versão com colisão do Pong, implemente o “efeito” sobre a bolinha. Isso pode ser feito da seguinte maneira: se a barra estiver subindo no momento do choque, a velocidade da bolinha no eixo y deverá ser acrescentada de um pequeno valor capaz de fazer com que ela suba ligeiramente mais rápido do que vinha fazendo. O mesmo ocorrerá se a barra estiver descendo.

E8.5 A função de colisão do Código 8.7 implementa qualquer tipo de colisão. No caso do Pong, a colisão é em geral mais simples, uma vez que a bolinha não irá colidir com o lado esquerdo da barra da esquerda nem com o lado direito da barra da direita. Escreva uma nova função de colisão simplificada para o Pong.

E8.6 Procure fazer uma análise sobre o Exercício E8.5 para ver quanto houve de melhora no novo código, buscando estimar o número de operações matemáticas e lógicas economizadas com a nova solução.

E8.6 Implemente um tratamento de exceção para a função load() do GameController do Pong de forma que, se ocorrer um erro ao carregar a imagem, a aplicação utilize uma imagem padrão.

E8.7 Dizemos que um objeto possui inteligência artificial quando realiza tarefas autônomas e independentes do jogador em si. Procure refazer a versão dos Códigos 8.16 e 8.17, fazendo com que as barras deixem de se mover aleatoriamente e procurem prever a posição em que a bolinha estará. Uma boa inteligência artificial deverá fazer uma previsão física da posição da bolinha, levando em conta sua velocidade, posição e direção. Tome o cuidado para não trapacear, respeitando uma velocidade constante de movimento da barrinha.

E8.8 Crie um novo nível de dificuldade para o Pong, colocando duas bolinhas que se movem independentemente uma da outra.

E8.9 Acrescente ao Exercício E8.8 a possibilidade de as duas bolinhas se chocarem. Neste caso, devem se comportar tal como quando se chocam com uma parede ou com a barrinha.

E8.10 Utilizando a escrita no modo gráfico, acrescente ao exercício anterior um placar. Como você apenas irá colocar o comando de entrada para o jogador no capítulo seguinte, neste momento o placar deverá apenas mostrar o número de vezes que a bolinha se chocou com cada parede lateral.

E8.11 Utilizando algum programa de desenho digital, crie uma tira de *sprite* animado para a bolinha, de forma que quando ela se move, realiza algum movimento.

E8.12 Investigue mais a fundo as classes de exceção e, baseado nelas, desenvolva dois outros casos de exceção para o javaPlay: resolução do monitor não suportada e posição de *sprite* inválida na tela.

Capítulo

9

Programação Guiada a Eventos

O entendimento mais comum de construção e execução de um programa talvez ainda seja similar ao que se observa em uma fábrica de automóveis tradicional, com a diferença de que, em vez de utilizarmos chapas de aço, porcas e parafusos como matéria-prima, utilizamos dados e, em vez de martelos e chaves de fenda, utilizamos métodos para transformar e recombinar a matéria-prima até conseguir o resultado desejado.

Essa forma de entender o funcionamento, e consequentemente a melhor abordagem para projetar e construir programas, é denominada *programação guiada a fluxo de dados*, pelo mesmo motivo que uma fábrica de automóveis poderia ser denominada manufatura guiada a fluxo de materiais. Ela é bastante útil para resolver diversos problemas computacionais, mas não é a única forma de se considerar o uso e a construção de programas.

Alternativamente, consideremos o método de trabalho encontrado em uma central de gerenciamento de emergências urbanas, comumente encontrada em grandes cidades no Brasil e no mundo, que acompanham eventos que porventura ocorram no cenário urbano – acidentes de trânsito, enchentes, incêndios, grandes congestionamentos etc. – e, quando necessário, acionam pessoal especializado (brigadas de incêndio, engenheiros de tráfego, equipes de paramédicos) para tomar as providências cabíveis. Nesse caso, em

vez de receber requisições e efetuar sequências de atividades para atender a elas, a central de gerenciamento de emergências deve monitorar um conjunto de eventos relevantes e manter, para cada possível ocorrência de cada evento relevante, quais processos devem ser disparados como resposta. Os eventos relevantes podem ocorrer a qualquer momento e em qualquer ordem, e a central de gerenciamento deve responder aos eventos à medida que eles ocorram.

A construção e utilização de programas pode se fundamentar no modelo de atividades da central de gerenciamento de emergências, em vez do modelo da fábrica de automóveis. Nesse caso, é preciso contar com um componente de software que monitore permanentemente a ocorrência de eventos relevantes, e que contenha um "catálogo" atualizado de ações correspondentes a cada evento relevante. À medida que os eventos ocorrerem e forem percebidos pelo componente de monitoramento, as ações correspondentes serão disparadas.

Conforme deve ser percebido intuitivamente, esse modelo de programação pode ser especialmente útil para a construção de sistemas interativos em que, com base em uma interface, os usuários possam informar que operações eles desejam que sejam efetuadas, na ordem e com os parâmetros que lhe forem mais convenientes. Essa é precisamente a situação encontrada na utilização da grande maioria dos jogos de computador. Portanto, essa forma de construir e utilizar programas – denominada *programação guiada a eventos* – é bastante útil e frequentemente utilizada no projeto e na construção de jogos de computador.

Neste capítulo estudaremos a programação guiada a eventos. Para isso, utilizaremos o motor de jogos *javaPlay*, que acompanha este livro, como implementação pré-fabricada de um monitor de eventos. Utilizaremos, também, recursos presentes na própria linguagem Java que facilitam a programação guiada a eventos. Apresentaremos recursos para a programação guiada a eventos de forma incremental, iniciando com exemplos simples e acrescentando, passo a passo, novos recursos, conceitos e técnicas.

9.1. Eventos e ações simples

Nosso exemplo inicial permitirá simplesmente a apresentação de uma tela estática. Esse será nosso ponto de partida, pois nos permitirá considerar um mesmo programa do ponto de vista de fluxo de dados e do ponto de vista de gerenciamento de eventos.

A programação guiada a fluxo de dados indicaria, para um programa com essa finalidade, a construção de um programa que pudesse ser acionado para, em seguida, apresentar a imagem correspondente à sua tela estática e, por exemplo, completar e encerrar sua atividade após um intervalo de tempo predeterminado.

Um programa construído dessa maneira, entretanto, não interage com o usuário, exceto pelo fato de que supostamente este seria quem o acionaria. Para tornar o programa mais interativo, uma possibilidade é oferecer ao usuário o controle que define quando a tela estática deve desaparecer e o programa deve encerrar sua atividade. Para isso, o programa já deverá monitorar a ocorrência de um evento, informado de alguma maneira pelo usuário, que indique o momento de encerrar sua atividade.

Nos capítulos anteriores, vimos o motor para controle de jogos *JavaPlay* que acompanha este livro e como utilizá-lo. O exemplo inicial deste capítulo, na verdade, é mais simples que os programas vistos anteriormente. Utilizaremos, também, recursos gráficos pré-construídos que acompanham o *JavaPlay*, para construir nossas ilustrações.

Nosso programa será composto por duas classes: a classe `Main`, como sempre, será o ponto de partida para o funcionamento do programa. Ela deverá conter um comando para definir o que deve acontecer quando o programa for acionado e um comando para promover efetivamente esse acionamento.

A segunda classe que deverá estar presente em nosso programa deverá expor a tela estática prevista no programa, e a partir daí passar a monitorar a ocorrência de eventos relevantes para, como consequência, disparar os processos correspondentes e gerar os efeitos desejados. Em nosso caso, existirá somente um evento relevante, que será o acionamento do botão esquerdo do mouse com o cursor apontando para uma determinada região da tela. O processo (também único) correspondente a esse evento será o encerramento do programa.

A linguagem Java contém objetos predefinidos que permitem construir com facilidade monitores para os eventos que mais usualmente são utilizados a fim de direcionar o comportamento de programas, como, por exemplo, eventos relacionados ao acionamento de dispositivos de comunicação com o computador (mouse, teclado etc.). O acesso a esses objetos predefinidos foi facilitado ainda mais no motor *JavaPlay*, que contém classes pré-fabricadas para simplificar o monitoramento desses dispositivos. No caso específico do

mouse, temos uma classe `Mouse` que facilita o monitoramento do acionamento desse dispositivo, a qual contém métodos como, por exemplo, `isLeftButtonPressed()`, que devolve o valor `TRUE` quando o monitor detecta o acionamento do botão esquerdo do mouse, e `getMousePos()`, que fornece as coordenadas cartesianas do ponto em que se encontra o cursor, ou seja, monitora a movimentação do mouse.

Dessa forma, temos já pré-construídos os recursos necessários para monitorar eventos (esses recursos já estão pré-construídos em Java; o motor *JavaPlay* apenas simplifica um pouco o acesso aos recursos da linguagem Java), e o que nos resta fazer é associar, na classe que construirmos, os processos referentes aos eventos relevantes que tivermos.

No Código 9.1 encontramos a classe `Main` descrita anteriormente, que aciona a classe `StartMenuController`. No Código 9.2 encontramos a classe `StartMenuController`, que utiliza um recurso gráfico pré-construído como tela estática a ser apresentada, constrói monitores para eventos relevantes produzidos pelo mouse e define os processos relativos aos eventos relevantes construídos.

Código 9.1 A Classe Main Para Nosso Primeiro Programa Guiado a Eventos

```
package demo;

import javaPlay.GameEngine;

public class Main
{
   public static void main(String[] args)
   {
      GameEngine.getInstance().addGameStateController
      (Global.0, new StartMenuController());
      GameEngine.getInstance().setStartingGameStateController
      (Global.0);
      GameEngine.getInstance().run();
   }
}
```

Na classe `Main`, utilizamos recursos da classe `GameEngine`, presente no motor *JavaPlay*. Os recursos utilizados são o método `addGameStateController`, que insere no programa o acesso a uma classe – em nosso caso, a classe `StartMenuController`; o método `setStartingGameStateController`, que indica qual classe, dentre as inseridas, deve ser acionada inicialmente; e a classe `run`, que efetivamente "dispara" o funcionamento do jogo.

A partir desse ponto, o controle está com a classe `StartMenuController`.

Essa classe implementa a interface `GameStateController`, presente no motor *JavaPlay*, e dessa forma expõe todos os métodos previstos naquela interface. Os nomes desses métodos são relativamente autoexplicativos, mesmo assim detalharemos cada método à medida que eles forem utilizados.

Em nosso exemplo inicial, fazemos uso de três métodos:

- `load`, que efetua os procedimentos iniciais da classe – em nosso caso, seleciona o recurso gráfico a ser apresentado na tela estática correspondente ao programa. Como esse método depende da disponibilidade do recurso gráfico para ser apresentado, ou seja, depende de um recurso externo ao programa para ser executado, é conveniente considerar a possibilidade de, por algum motivo externo, o recurso não estar lá. Por esse motivo, é utilizada a estrutura `try/catch` da linguagem Java, que tenta efetuar uma operação (bloco `try`) – no caso, a carga do recurso gráfico externo – mas, na eventualidade de ocorrer uma falha, executa um bloco alternativo (o bloco de captura da falha, ou seja, o bloco `catch`).
- `draw`, que apresenta na tela do computador o recurso gráfico selecionado para constituir a tela estática de nosso exemplo.
- `step`, que contém os monitores de eventos e consulta esses monitores a cada passo do relógio (o que explica o nome desse método – *step*). Em nosso exemplo simples, esse método monitora se o cursor do mouse foi movimentado para uma área específica e se o seu botão esquerdo foi acionado. Caso esses dois eventos ocorram simultaneamente, é acionado um método pré-fabricado que encerra a execução do programa.

Construímos, dessa forma, nosso primeiro programa totalmente guiado a eventos. O programa nos apresenta uma tela estática e monitora a ocorrência de um evento relevante, proveniente do acionamento do mouse, que tem o poder de provocar o disparo da ação que encerra as operações do programa.

9.2. Eventos de teclado

Em vez de encerrar as operações do programa, o acionamento do mouse pode provocar o disparo de eventos mais elaborados. Por exemplo, esse evento pode indicar que o jogo passou para um novo estado – uma nova fase, por exemplo – em que novos eventos passem a ser os eventos relevantes, ou que o significado dos mesmos eventos monitorados anteriormente seja atualizado.

Essa condição pode ser incluída no exemplo da seção anterior, pelo acréscimo de uma nova classe controladora de eventos (além da classe já presente `StartMenuController`), e de uma referência a essa classe dentro da classe `Main`.

Nosso jogo passará, portanto, a ter dois conjuntos de monitores de eventos, e um evento específico para provocar a transição de um conjunto para outro. No segundo conjunto de monitores de eventos, incluiremos um *personagem*, que será uma animação controlada por comandos do teclado. Em outras palavras, teremos um componente do jogo cuja apresentação gráfica será movimentada como consequência de ações que serão disparadas quando determinados eventos relevantes ocorrerem.

Faremos uso, mais uma vez, de recursos gráficos que acompanham o motor *JavaPlay*. Naturalmente, esses recursos poderão ser alterados ou trocados por novos recursos que você venha a construir.

Assim como a classe `StartMenuController`, a nova classe controladora de eventos implementará a interface `GameStateController`. No roteiro de nosso jogo teremos eventos internos a um conjunto de eventos, cujas ações manterão o controle do jogo dentro de uma classe controladora, e eventos de transição, que terão como consequência a delegação do controle de uma classe controladora para outra. Por enquanto, teremos apenas duas classes controladoras, embora possamos ter, em um jogo mais elaborado, diversas classes e uma estrutura de transições bastante elaborada.

Código 9.2 A Classe StartMenuController Para Nosso Primeiro Programa Guiado a Eventos

```
package demo;

import java.awt.Graphics;
import java.awt.Point;
import java.util.logging.Level;
import java.util.logging.Logger;
import javaPlay.GameEngine;
import javaPlay.GameStateController;
import javaPlay.Mouse;
import javaPlay.Sprite;

public class StartMenuController implements
GameStateController
{
   private Sprite menu;

   public void load()
   {
      try
      {
```

Código 9.2 A Classe StartMenuController Para Nosso Primeiro Programa Guiado a Eventos (cont.)

```
            menu = new Sprite("startmenu.jpg", 1, 800, 600);
         }
         catch (Exception ex)
         {
            Logger.getLogger(StartMenuController.class.getName())
            .log(Level.SEVERE, null, ex);
         }
      }

      public void unload()
      {
      }

      public void start()
      {
      }

      public void step(long timeElapsed)
      {
         Mouse m = GameEngine.getInstance().getMouse();
         if(m.isLeftButtonPressed() == true)
         {
         Point p = m.getMousePos();
         if((p.x >= 363) && (p.y >= 353) && (p.x <= 485) && (p.y <= 389))
         {
            GameEngine.getInstance().requestShutdown();
         }
      }
      }

      public void draw(Graphics g)
      {
         menu.draw(g, 0, 0);
      }

      public void stop()
      {
      }
   }
```

Nossa nova classe `Main` será parecida com aquela apresentada no Código 9.1. A única diferença é que teremos duas classes controladoras disponíveis em nosso jogo, em vez de apenas uma. Essa classe `Main` está apresentada no Código 9.3.

A nova classe `StartMenuController` também será parecida com aquela apresentada na Código 9.2. A diferença é que, em vez de o evento relevante de acionamento do mouse encerrar a execução do jogo, esse mesmo evento delegará o controle do jogo para a nova classe controladora. Essa nova classe `StartMenuController` está apresentada no Código 9.4.

A classe controladora que incluiremos nesta seção será denominada `SimpleController`. Ela cuidará dos eventos de teclado que controlarão os movimentos do personagem do jogo. Nosso "jogo" simples será constituído apenas por um personagem que conseguiremos movimentar. Essa classe está apresentada no Código 9.5. Em seu conteúdo podemos identificar a associação de eventos de teclado a ações do personagem, bem como a associação de um evento específico de mouse ao reinício do jogo. Podemos identificar, também, a carga dos recursos gráficos pré-fabricados apropriados.

Como incluiremos um personagem, precisaremos também de uma classe que defina as ações possíveis para ele. Essa classe será referenciada e utilizada a partir da classe `SimpleController`, já que `SimpleController` será a classe responsável por determinar as ações do personagem, como pode ser visto no Código 9.6. Podemos identificar, nessa classe, a definição do que deve ocorrer no caso de cada ação possível do personagem ser ativada. É interessante observar, nessa classe, como cada imagem dentro do modelo de representação gráfica do personagem é acionada para ser apresentada no momento apropriado correspondente a cada ação.

Código 9.3 A Classe Main Para o Programa com Eventos de Teclado

```
package demo;

import javaPlay.GameEngine;

public class Main
{
   public static void main(String[] args)
   {
      GameEngine.getInstance().addGameStateController
      (Global.0, new StartMenuController());
      GameEngine.getInstance().addGameStateController
      (Global.1, new SimpleController());
      GameEngine.getInstance().setStartingGameStateController
      (Global.0);
      GameEngine.getInstance().run();
   }
}
```

9.3. Eventos definidos no contexto do jogo

Se você implementou e testou os programas até aqui, deve ter notado que a interatividade de nosso jogo ainda deixa bastante a desejar. Considerando que uma das principais vantagens da programação guiada a eventos é

melhorar a interatividade, isso indica que ainda precisamos melhorar nosso monitoramento e gerenciamento de eventos.

Até aqui, construímos eventos e ações diretamente relacionadas com ações do usuário. Para completar as possibilidades necessárias para a construção de um jogo, é preciso acrescentar eventos e ações relacionadas com o contexto do próprio jogo.

Por exemplo, no programa construído até a seção anterior, apesar de termos, na tela de fundo `scene.scn` exposta pela classe `SimpleController` o desenho de paredes externas que deveriam delimitar o movimento de nosso personagem, essas paredes são ignoradas pelas ações de movimentação do personagem.

Nesta seção incluiremos recursos em nossas classes já existentes para delimitar o movimento do personagem para dentro da área da tela desenhada em `scene.scn`. Incluiremos, também, um "alvo" de modo a encerrar o jogo: quando conseguirmos posicionar o personagem sobre o alvo, esse evento será detectado, e a ação correspondente – encerramento da execução do programa – será efetuada.

Todos esses eventos poderão ocorrer no estado do jogo correspondente ao que é controlado pela classe `SimpleController`. Portanto, essa será a classe modificada. As outras classes não precisarão sofrer alterações, uma vez que os eventos e as ações correspondentes a elas não serão modificados.

A classe `SimpleController`, modificada para monitorar eventos definidos no contexto do jogo, está apresentada no Código 9.7.

Código 9.4 A Classe StartMenuController para o programa com eventos de teclado

```
package demo;

import java.awt.Graphics;
import java.awt.Point;
import java.util.logging.Level;
import java.util.logging.Logger;
import javaPlay.GameEngine;
import javaPlay.GameStateController;
import javaPlay.Mouse;
import javaPlay.Sprite;

public class StartMenuController implements GameStateController
{
   private Sprite menu;

```

Código 9.4 A Classe StartMenuController para o programa com eventos de teclado (cont.)

```
   public void load()
   {
      try
      {
         menu = new Sprite("startmenu.jpg", 1, 800, 600);
      }
      catch (Exception ex)
      {
         Logger.getLogger(StartMenuController.class.getName())
         .log(Level.SEVERE, null, ex);
      }
   }

   public void unload()
   {
   }

   public void start()
   {
   }

   public void step(long timeElapsed)
   {
      Mouse m = GameEngine.getInstance().getMouse();

      if(m.isLeftButtonPressed() == true)
      {
         Point p = m.getMousePos();

         if((p.x >= 363) && (p.y >= 353) && (p.x <= 485) && (p.y <= 389))
         {
            GameEngine.getInstance().setNextGameStateController
            (Global.1);
         }
      }
   }

   public void draw(Graphics g)
   {
      menu.draw(g, 0, 0);
   }

   public void stop()
   {
   }
}
```

Código 9.5 A Classe SimpleController para o programa com eventos de teclado

```
package demo;

import java.awt.Graphics;
import java.awt.Point;
import java.util.logging.Level;
import java.util.logging.Logger;
import javaPlay.GameEngine;
import javaPlay.GameStateController;
import javaPlay.Keyboard;
```

Código 9.5 A Classe SimpleController para o programa com eventos de teclado (cont.)

```
import javaPlay.Mouse;
import javaPlay.Scene;
import javaPlay.Sprite;

public class SimpleController implements GameStateController
{
   private Sprite playerSprite;
   private Player player;
   private Scene scene;

   public void load()
   {
      try
      {
         playerSprite = new Sprite(“tankplayer.png”, 8, 32, 32);
         scene = new Scene();
         scene.loadFromFile(“scene.scn”);
         player = new Player();
         player.setSprite(playerSprite);
         scene.addOverlay(player);
      }
      catch (Exception ex)
      {
         Logger.getLogger(SimpleController.class.getName()).
         .log(Level.SEVERE, null, ex);
      }
   }

   public void unload()
   {
   }

   public void start()
   {
      player.x = 92;
      player.y = 114;
   }

   public void step(long timeElapsed)
   {
      Keyboard k = GameEngine.getInstance().getKeyboard();

      if(k.keyDown(Keyboard.UP_KEY) == true)
      {
         player.moveUp();
      }
      if(k.keyDown(Keyboard.DOWN_KEY) == true)
      {
         player.moveDown();
      }
      if(k.keyDown(Keyboard.LEFT_KEY) == true)
      {
         player.moveLeft();
      }
      if(k.keyDown(Keyboard.RIGHT_KEY) == true)
      {
         player.moveRight();
      }
      Mouse m = GameEngine.getInstance().getMouse();
```

Código 9.5 A Classe SimpleController para o programa com eventos de teclado (cont.)

```

         if(m.isLeftButtonPressed() == true)
         {
            Point p = m.getMousePos();
            if((p.x >= 363) && (p.y >= 353) && (p.x <= 485) && (p.y <= 389))
            {
               GameEngine.getInstance().setNextGameStateController
               (Global.1);
            }
      }
         player.step(timeElapsed);
      }

      public void draw(Graphics g)
      {
         scene.draw(g);
      }

      public void stop()
      {

      }
}
```

Código 9.6 A Classe Player que controla o personagem no programa com eventos de teclado

```
package demo;

import java.awt.Graphics;
import javaPlay.GameObject;
import javaPlay.Sprite;

public class Player extends GameObject
{
   private Sprite sprite;
   private int currFrame;

   private final int UP_FRAMES = 0;
   private final int LEFT_FRAMES = 2;
   private final int RIGHT_FRAMES = 6;
   private final int DOWN_FRAMES = 4;

   public void setSprite(Sprite sprite)
   {
      this.sprite = sprite;
      currFrame = 0;
   }

   public void moveUp()
   {
      y--;
      currFrame = UP_FRAMES;
   }

   public void moveDown()
   {
      y++;
```

Código 9.6 A Classe Player que controla o personagem no programa com eventos de teclado (cont.)

```
            currFrame = DOWN_FRAMES;
        }
            public void moveLeft()
        {
            x--;
            currFrame = LEFT_FRAMES;
        }

        public void moveRight()
        {
            x++;
            currFrame = RIGHT_FRAMES;
        }

        public void step(long timeElapsed)
        {
        }

        public void draw(Graphics g)
        {
            sprite.setCurrAnimFrame(currFrame);
            sprite.draw(g, x, y);
        }
    }
```

Código 9.7 A Classe SimpleController com eventos no contexto do jogo

```
    package demo;

    import java.awt.Graphics;
    import java.awt.Point;
    import java.util.logging.Level;
    import java.util.logging.Logger;
    import javaPlay.GameEngine;
    import javaPlay.GameStateController;
    import javaPlay.Keyboard;
    import javaPlay.Mouse;
    import javaPlay.Scene;
    import javaPlay.Sprite;

    public class SimpleController implements GameStateController
    {
        private Sprite playerSprite;
        private Player player;
        private Scene scene;

        public void load()
        {
            try
            {
                playerSprite = new Sprite("tankplayer.png", 8, 32, 32);

                scene = new Scene();
                scene.loadFromFile("scene.scn");

                player = new Player();
                player.setSprite(playerSprite);

                scene.addOverlay(player);
```

Código 9.7 A Classe SimpleController com eventos no contexto do jogo (Cont.)

```
      }
      catch (Exception ex)
      {
         Logger.getLogger(SimpleController.class.getName()).
         .log(Level.SEVERE, null, ex);
      }
   }

   public void unload()
   {
   }

   public void start()
   {
      player.x = 92;
      player.y = 114;
      player.moveRight();
   }

   public void step(long timeElapsed)
   {
      Keyboard k = GameEngine.getInstance().getKeyboard();

      if(k.keyDown(Keyboard.UP_KEY) == true)
      {
         player.moveUp();
      }
      if(k.keyDown(Keyboard.DOWN_KEY) == true)
      {
         player.moveDown();
      }
      if(k.keyDown(Keyboard.LEFT_KEY) == true)
      {
         player.moveLeft();
      }
      if(k.keyDown(Keyboard.RIGHT_KEY) == true)
      {
         player.moveRight();
      }
         checkCollision();
      checkFinalization();

      Mouse m = GameEngine.getInstance().getMouse();

      if(m.isLeftButtonPressed() == true)
      {
         Point p = m.getMousePos();

         if((p.x >= 363) && (p.y >= 353) && (p.x <= 485) &&
            (p.y <= 389))
         {
            GameEngine.getInstance().setNextGameStateController
            (Global.1);
         }
      }

      player.step(timeElapsed);
   }

   public void checkCollision()
```

Código 9.7 A Classe SimpleController com eventos no contexto do jogo (Cont.)

```
      {
         if(player.y < 64)
         {
            player.moveDown();
         }
         if(player.y > 448)
         {
            player.moveUp();
         }
         if(player.x < 64)
         {
            player.moveRight();
         }
         if(player.x > 672)
         {
            player.moveLeft();
         }
      }

      public void checkFinalization()
      {
         if((player.y > 416) && (player.y <= 448) && (player.x > 640)
         && (player.x <= 672))
         {
            GameEngine.getInstance().requestShutdown();
         }
      }
      public void draw(Graphics g)
      {
         scene.draw(g);
      }

      public void stop()
      {
      }
   }
```

EXERCÍCIOS

E9.1 No programa do Código 9.5, é preciso posicionar o cursor em um ponto específico da tela para poder reiniciar o jogo. Esse ponto não tem um significado claro no desenho da tela `scene.scn`. Modifique o programa para que o evento relevante associado à ação de reiniciar o jogo corresponda a apertar o botão esquerdo do mouse, independentemente de onde esteja o cursor.

E9.2 No programa do Código 9.6 são utilizadas quatro constantes, denominadas `UP_FRAMES`, `LEFT_FRAMES`, `RIGHT_FRAMES` e `DOWN_FRAMES`. Explique o que significam e para que servem essas constantes.

E9.3 No programa do Código 9.7 o personagem do jogo tem seus movimentos delimitados para dentro das paredes externas da tela `scene.scn`, mas as paredes internas no desenho dessa tela não são reconhecidas. Modifique esse programa para que as paredes internas também sejam reconhecidas e o personagem não

possa mais "atravessá-las". Você consegue generalizar o seu programa, para que o conceito de colisão com paredes se torne um método genérico?

E9.4 No programa do Código 9.7, quando o personagem atinge o alvo, o jogo é encerrado abruptamente. Modifique o programa para que, em vez de encerrar o jogo, o alvo indique um evento de transição para um novo conjunto de eventos, por exemplo, a apresentação de uma tela estática com uma mensagem de fim de jogo.

E9.5 Modifique o programa do Código 9.7, incluindo um novo personagem que faça o papel de "inimigo": quando o seu personagem colidir com o inimigo, o jogo se encerrará. Você consegue fazer o inimigo se movimentar por conta própria?

A

Desenvolvimento e Design de Jogos

A.1. Introdução

O objetivo deste livro é fazer com que você aprenda a programar. Escolhemos para os casos práticos o tema de jogos. Apesar de os projetos desenvolvidos até o momento serem relativamente simples, é importante frisar que foi apresentada a você uma arquitetura de um motor de jogo semelhante à grande maioria dos motores profissionais usados pela indústria. Uma vez que você se sinta confortável com os conceitos básicos de programação e de Java, será simples avançar com passos maiores e desenvolver jogos mais complexos.

Este apêndice apresentará resumidamente o processo de desenvolvimento e design de jogos maiores, fornecendo ponteiros e caminhos a seguir.

Um jogo é um software especial, pois contém elementos muito variados: módulos de computação gráfica, inteligência artificial, redes, multimídia etc. Todos esses módulos devem funcionar em perfeita harmonia, obedecendo a uma característica fundamental sob o ponto de vista técnico: deve ser um software que funciona em tempo real, portanto, eficiente e rápido. Para que isso seja possível é necessário explorar ao máximo o hardware dedicado, e é fundamental que o jogo esteja baseado sobre diversas APIs, tais como o OpenGL, DirectX, OpenAL etc. Chamamos de APIs as Application Programming Interface ou Interface de Programação de Aplicativos. Essas camadas de

software se responsabilizarão por resolver acesso ao hardware gráfico, à placa de som e a outros recursos de hardware.

Além disso, enquanto a maioria dos softwares precisa apenas seguir uma série de requisitos e cumprir bem os propósitos para os quais foram elaborados, uma característica imprescindível para um jogo é que ele deve ser divertido e agradável de se utilizar, uma vez que a sua principal razão de ser é proporcionar entretenimento para o seu usuário.

Todas essas características, somadas ao fato de que um jogo necessita sempre lançar mão do "estado da arte" em termos computacionais, faz com que o desenvolvimento de jogos seja uma área fascinante, cheia de desafios e "magias".

A.2. Design de um jogo

Se um carro não anda, não serve para muita coisa, pois não cumpre seu papel principal, mesmo que o farol acenda, o rádio funcione e a buzina toque.

Na indústria de software, chamamos as principais especificações de um software de requisitos. Tal como no exemplo do carro, se um sistema não cumpre os seus principais requisitos, não serve para muita coisa. Assim, um banco de dados tem como seus principais requisitos armazenar e buscar dados de forma eficiente. Portanto, por melhor que seja sua interface, eficiência ou segurança, se ele não cumprir o requisito principal, que é armazenar e gerir dados, está fadado a ser descartado.

Um jogo tem como seu principal requisito ser divertido. Se o jogo não entretiver o usuário, não está cumprindo seu principal propósito. O grande problema é que o fator diversão não é algo tão objetivo, como armazenar dados ou andar para frente.

Sempre que se pretende criar um jogo, o projetista deve antes pensar qual é o fator de diversão que ele possui. Há uma área do conhecimento humano que procura estudar a ludicidade no ser humano: o que provoca diversão, como medi-la e que elementos comuns os jogos possuem, capazes de garantir essa diversão. Há diversos livros muito interessantes sobre o tema, sendo que *Homo Ludens – O jogo como elemento da cultura* [20] é uma referência obrigatória e *Theory of Fun for Game Design* [21] é uma excelente adaptação das ideias para o campo dos videogames. O estudo da ludicidade vai além das fronteiras do entretenimento digital, e os psicólogos afirmam categoricamente

que a diversão é uma necessidade de qualquer ser humano: gostamos de viajar, apreciar um bom filme, sair com amigos para comer uns petiscos, assistir a um jogo de futebol, contar e ouvir coisas divertidas... Há inclusive quem diga que a vida do ser humano se restringe a trabalhar, descansar e entreter-se.

Dentro do escopo de querer entender o que é o lúdico, a pergunta central é: "por que algumas coisas nos divertem e outras não?". Apesar de ser uma pergunta difícil e nem sempre existir uma resposta objetiva, se quisermos desenvolver um produto que seja divertido, devemos pelo menos saber como começar a responder essa pergunta.

Sob o ponto de vista fisiológico, sabe-se que alguns elementos químicos produzidos pelo corpo humano podem gerar uma sensação de bem-estar, tal como ocorre com a endorfina (*endo* = interno, *morfina* = analgéstico), dando ao cérebro uma sensação de euforia. Este e outros hormônios, tal como a adrenalina e o cortisol, são produzidos naturalmente pelo corpo humano em diversas situações. Obviamente quando a sua produção é muito alta, pode produzir efeitos desagradáveis a uma pessoa, mas importantes para sua sobrevivência, tal como ocorre num susto ou numa situação de perigo. Porém, quando são produzidos numa medida pequena e de forma natural, portanto, sem o uso de remédios ou drogas, podem produzir uma sensação de bem-estar saudável. Sabe-se que esses neurotransmissores são produzidos quando uma pessoa pratica esporte, quando exercita sua memória e quando o cérebro recebe desafios "controlados", dentre outras situações. Por desafios "controlados" entendemos um desafio que não é muito banal, tampouco exige um esforço extraordinário do cérebro, como pode ocorrer em situações em que há situação de perigo real. Obviamente quando o desafio nos pressiona para uma situação de risco, os hormônios serão produzidos, mas sairão da margem do "bem-estar". Não estamos sendo reducionistas, dizendo que apenas o desafio é o que nos causa diversão: ajudar uma pessoa, ouvir uma boa música, comer um delicioso sorvete ou participar de uma boa conversa entre amigos também podem causar bem-estar e não são situações relacionadas a estar vencendo desafios.

No caso dos videogames, é justamente esse fator desafio que importa: criar desafios para o cérebro, porém em situações totalmente desconectadas da realidade, portanto, dando-nos uma total sensação de segurança, caso não consigamos "vencer". É por isso que passamos muito mal numa situação de assalto à mão armada real, mas podemos nos divertir bastante ao participar

de um combate virtual. Poder perder é uma necessidade de um jogo, pois se sempre ganharmos não existe desafio, ou pelo menos ele pode se tornar muito pequeno a ponto de não exercitar nosso cérebro o suficiente. Jogar futebol pode ser muito divertido, mas ficar passando a bola um para outro, sem nenhum propósito, pode se tornar uma tarefa sem graça.

Dessa forma, redirecionamos a discussão do que é o lúdico ou o que é a diversão para tentar responder quais e como devem ser os desafios a serem criados. A lista a seguir relaciona alguns dos principais elementos de desafio de um jogo:

- Corrida: movimentar-se para estar em primeiro lugar
- Combate: confrontar-se com entidades adversárias
- Comando: dar ordens e querer ser correspondido
- Construir: utilizar elementos básicos para construir algo maior
- Colecionar: procurar juntar elementos de interesse
- Negociar: trocar elementos menos importantes por outros mais importantes
- Conectar: montar uma ideia ou um elemento, partindo de dois ou mais fatos ou elementos
- Escapar: evitar a presença de algum elemento indesejável ou prejudicial
- Encontrar: buscar um elemento desejado e não-visível
- Esconder: evitar ser visto ou encontrado por um elemento indesejado
- Aprender: buscar algum conceito importante e que não se possui

Além de poder categorizar os jogos pelo tipo de desafio imposto, é possível subdividi-los em gêneros ou estilos. Cada um desses gêneros poderá explorar mais de um dos desafios listados. Alguns dos gêneros transcendem o aspecto do desafio em si e referem-se a um estilo de design. Conhecer os gêneros de jogos também é importante para o processo de design, uma vez que esse conhecimento irá nortear o tipo de desafio, bem como o estilo que um projeto deverá seguir. Assim, os principais gêneros de jogos são:

- Ação: são jogos que levam o jogador a combater e realizar ações que o ajudem a vencer e atingir objetivos específicos. Counter Strike, Halo e Spacewar são alguns exemplos.
- Aventura: Jogos que levam o jogador a encontrar elementos, escapar de situações e colecionar itens. The Legend of Zelda e Zork são exemplos destes jogos.

- Esporte: Jogos que procuram recriar as mesmas regras de um esporte real. Fifa Soccer e Wii Sports são alguns.
- Estratégia de Tempo Real: também conhecidos como RTS (Real Time Strategy), permitem que o jogador crie estratégias, negocie, comande e construa. Muitos jogos deste gênero costumam explorar uma estratégia de câmera denominada de Gods View (visão de Deus), de forma a dar uma visão ampla do mundo ao jogador. Warcraft e Command and Conquest são exemplos clássicos.
- Educacionais: Jogos elaborados com o propósito de ensinar algum conceito. Estes jogos acabam seguindo algum dos outros gêneros que estão listados.
- Estratégia em Turnos: também conhecidos como Turn-based Strategy, são parecidos ao de estratégia em tempo real, porém cada jogador deve esperar por seu turno para efetuar uma jogada. O xadrez e Civilization são exemplos.
- Luta: Jogos tipicamente de combate, podendo ser desde lutas marciais a lutas com armas. Mortal Kombat e Tekken são alguns.
- Jogos de Primeira Pessoa: também conhecidos como FPS (First Person Shooters), são jogos em que o jogador é transposto para a posição da câmera. Muitos jogos de ação se mesclam a este gênero. Doom, Quake, Far Cry são bons exemplos.
- Massive Multiplayer Online Games (MMOG): Jogos em rede, explorando a interação com personagens reais. Ultima Online e Ragnarok são alguns dos mais populares.
- Música: Jogos que exploram o ritmo de música ou a manipulação de algum instrumento, tais como Dance, Dance Revolution e Guitar Hero.
- Plataforma: Esta categoria não se deve ao tipo de desafios que o compõe propriamente dito, porém a um estilo de design, em que personagens andam e correm por um cenário que é movido na tela, enquanto o personagem tende a ficar no centro da tela. *Super Mario Bros.* e *Pitfall* são alguns exemplos.
- *Role-Playing Games* (RPG): Jogos que criam enredos e estórias ao longo de sua execução. *Diablo* e *Final Fantasy* são alguns dos mais conhecidos.

- *Serius Games* ou jogos sérios: jogos que procuram simular desafios e situações reais referentes a um contexto, tendo muitas vezes como objetivo o treinamento ou a educação dentro de uma realidade. *Us Army* é um dos mais conhecidos, mas em geral são jogos feitos por encomenda, para uma empresa ou campanha específica.
- Simulação: jogos que procuram simular situações reais. Flight Simulator e SimCity são exemplos.
- Quebra-cabeças: Jogos que exploram a lógica e a resolução de problemas. Tetris e Bejeweled são alguns dos mais conhecidos.

Juntamente com os desafios, há um elemento que todos os jogos devem possuir: regras, sem as quais não existem jogos, nem mesmo os jogos esportivos ou jogos de tabuleiro. Assim, ao planejar um jogo, o desenvolvedor irá de alguma forma traduzir e escrever os desafios através das regras. As regras de um jogo definem um conceito muito importante nos videogames: a jogabilidade ou *gameplay*.

Assim como não faz sentido criar um filme sem antes ter um roteiro bem elaborado, também é impossível partir para o desenvolvimento de um jogo sem antes ter um documento sem o seu *gameplay.*

Antes de desenvolver um jogo, deve-se produzir um documento de design, que irá conter a descrição do *gameplay*, portanto, as regras em detalhes. Esse documento de design pode ser encarado como uma espécie de manual de instruções para os futuros desenvolvedores do jogo [15]. De fato, tão importante é esse documento, que o processo de desenvolvimento não pode começar sem que ele esteja pronto. Além das regras, o documento deverá conter outros elementos, como os listados a seguir. Entretanto, todos os demais elementos são de certa forma acessórios: pode haver jogos sem enredos, sem gráficos sofisticados e até sem música, porém nunca haverá um jogo sem regras e sem desafios.

- Roteiro: chamados roteiros interativos, no caso de jogos, pois diferentemente dos roteiros de filmes, devem ter espaço para interferência do usuário no desencadeamento da estória. Ao elaborar o roteiro deve-se ter em conta qual o estilo do jogo que se está desenvolvendo. É normal usar uma estória para contextualizar o jogo e justificar parte de suas regras.

- Game Art: Entende-se por *Game Art* a conceituação artística do jogo. Hoje em dia, dada a complexidade das estórias e dos cenários elaborados, é comum que esta parte do documento seja escrita por um artista. Dentro deste item deverão ser expostos quais as principais características dos cenários, esboços de personagens, descrição das texturas fundamentais, mapas e descrições das fases (também denominado de *level design*). O livro [5] descreve detalhadamente como é a elaboração deste tópico.

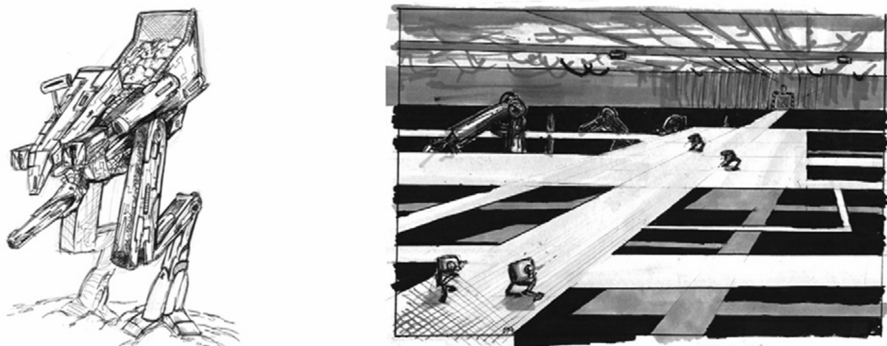


Figura A.1 Exemplo da conceituação artística de um personagem e de um cenário.

- *Gameplay*: Nesta parte do documento deve-se descrever como será a jogabilidade, como visto anteriormente. Nesta descrição deve ficar claro que o jogo é divertido e irá proporcionar desafios interessantes. Esta parte do documento é muito importante para guiar os programadores para grande parte da programação, especialmente no que se refere à implementação lógica do jogo.
- Interface: Pode-se dividir a interface em *ingame* e *outgame*. A primeira consiste na instrumentação disponível durante o jogo e é responsável pela entrada de dados do jogador para a aplicação. A interface *outgame* é a forma de apresentar a introdução do jogo, sua configuração, instruções etc. Os artistas gostam de dizer que a melhor interface é aquela que passa despercebida para o jogador, permitindo que ele possa focar no jogo em si, na história e nas ações. É dentro da interface *ingame* que devemos projetar os chamados HUDs (*Head Up Display*), que consistem em imagens que se sobrepõem a toda a cena, tais como miras, pontos, vidas, munição etc.

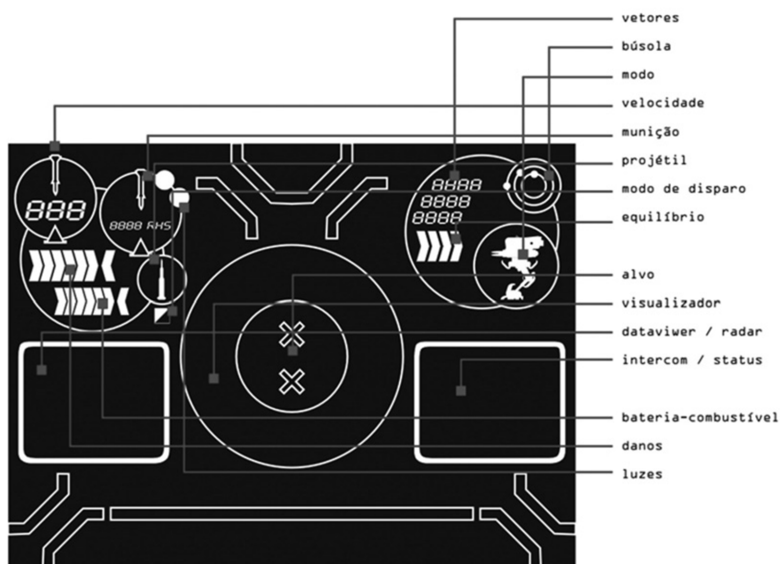


Figura A.2 Exemplo do design de uma interface *ingame.*

Terminada a etapa de conceituação, o desenvolvimento de um game divide-se em dois caminhos distintos: o de desenvolvimento artístico e o de programação, havendo entretanto, uma grande interseção entre ambas. A criação artística pode ser compreendida na elaboração dos *assets* do jogo, ou seja, os elementos que serão usados para sua montagem: modelos 3D, texturas, terrenos etc.

A.3. Desenvolvimento artístico

O desenvolvimento artístico também se divide em diversas etapas. Em projetos pequenos haverá uma sobreposição de responsabilidades entre os profissionais, porém em projetos grandes haverá uma distribuição específica de tarefas. A seguir descrevem-se as principais subáreas do desenvolvimento artístico.

Modelagem 3D

A equipe de modelagem 3D será responsável por criar os objetos tridimensionais e os cenários. A geometria de um jogo pode ser dividida em dois tipos: modelagem estrutural e modelagem de elementos dinâmicos. Essa diferenciação existe pelo fato de que os modelos estruturais, por não sofrerem alteração de posição, sofrerão um pré-processamento, de maneira a otimizar o processo de visualização. A modelagem estrutural consistirá basicamente na criação do cenário em si, o terreno e alguns outros elementos estáticos.

Para esta etapa os principais softwares utilizados são o 3DSMAX [7], MAYA [2], Softimage XSI [17] e Blender [12], sendo este último um software gratuito. Alguns recursos que esses programas oferecem, tornando-os ótimos para este processo são:

- Ferramentas de modelagem baseadas em polígonos: toda a modelagem deverá ser feita por polígonos. Assim sendo, é importante que haja uma forma fácil de manipulá-los.
- Ferramentas intuitivas para texturização: grande parte da riqueza de uma modelagem está na boa aplicação de texturas sobre os modelos. É comum, por exemplo, ter de aplicar um mapeamento de texturas para polígonos individuais.
- Boas ferramentas para otimização de polígonos: é comum, durante o processo de modelagem, criar objetos com mais polígonos do que se pode suportar no jogo. Assim sendo, é importante que um pacote de modelagem forneça recursos para reduzir o número de polígonos de objetos, minimizando a sua perda de qualidade.
- Boa interface de visualização: para o artista é importante que, à medida que um objeto é construído, possa-se acompanhar, em tempo real, como ele será visto no jogo.
- Fácil integração com motores de jogos: uma vez modelados os objetos ou a cena, esta será levada para um motor. Para tanto, é importante que os modeladores sejam capazes de exportar a geometria para formatos tipicamente usados por esses motores.

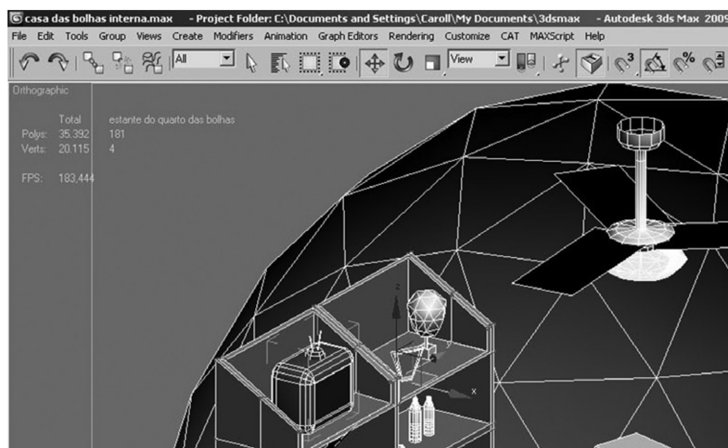


Figura A.3 As interfaces "*what you see is what you play*" permite que o artista possa ver em tempo real, na interface do software de modelagem, como seus modelos ficarão na renderização final do jogo. Imagens extraídas do 3DSMAX.

Neste processo, uma virtude importante que o artista deve ter é ser capaz de modelar objetos com o menor número de polígonos possível. Otimizando a modelagem, o cenário pode ser mais extenso e mais objetos podem ser inseridos nele.

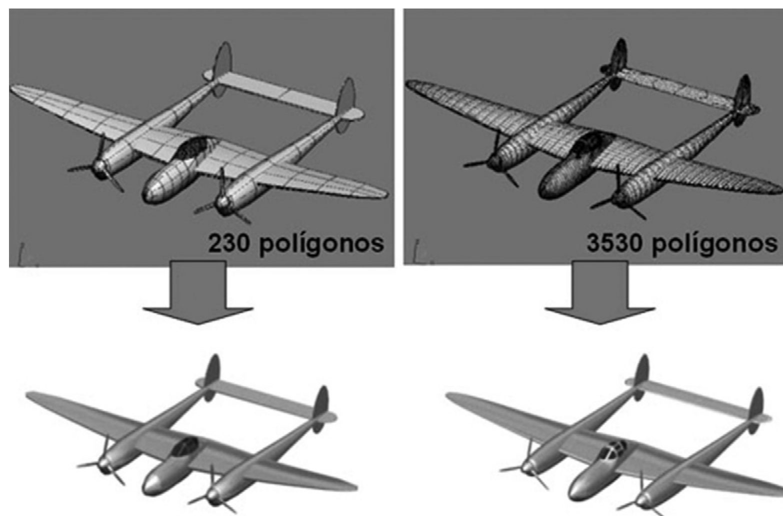
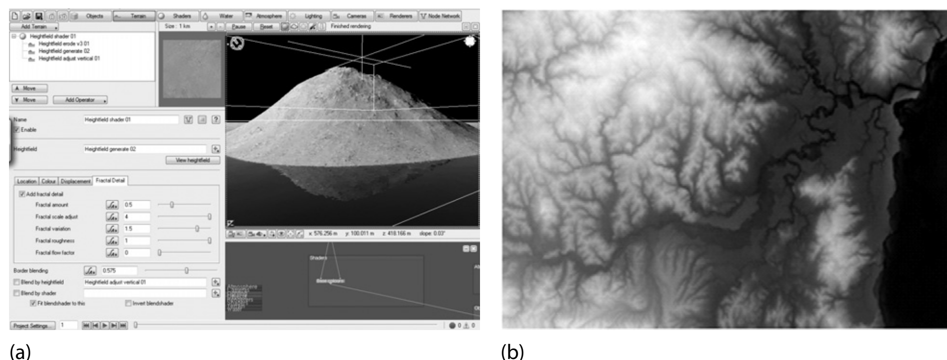


Figura A.4 Um mesmo modelo está representado em duas resoluções diferentes de polígonos. Perceba-se que o resultado final é bastante semelhante, embora o avião da direita possua 15 vezes mais polígonos que o da esquerda.

Terrenos

Os terrenos também são conhecidos como *heigh maps* e consistem em imagens com diversas tonalidades de cinzas. Os pixels escuros corresponderão a áreas baixas do terreno e os claros a partes mais altas. Assim sendo, elaborar um terreno resume-se a criar um mapa de altura adequado para o cenário, o qual pode ser pintado manualmente, usando algum software de edição de imagem. Para relevos mais complexos existem inúmeras ferramentas capazes de gerar mapas mais detalhados, bem como editá-los de forma mais intuitiva. Além de gerar os mapas de altura, esses softwares são capazes de gerar suas texturas correspondentes. Algumas das ferramentas mais utilizadas são o VueD'Espirit [10] e o Terragen [14]. Motores de jogos completos, tais como o Unity [4] e o Torque [13], também costumam possuir incorporados editores de terrenos, facilitando bastante o processo de integração.



(a) (b)

Figura A.5 (a) Software dedicado à criação de terrenos (extraído do software Terragen [14]); (b) Exemplo de um mapa de altura.

Os mapas de altura serão projetados sobre malhas regulares de polígonos. Dependendo do motor, isso será feito na ferramenta de modelagem 3D ou no próprio editor do motor.

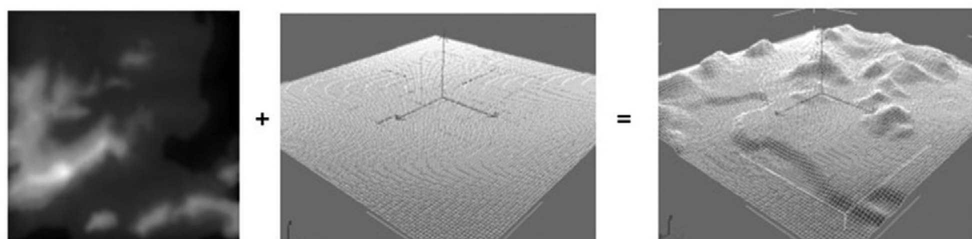


Figura A.6 Um mapa de altura é aplicado sobre uma malha de polígonos regulares, formando um terreno 3D.

A.4. Programação/Motores de jogos

Um motor de um carro é responsável por fazê-lo andar. Ao dar a ignição do veículo, o motorista coloca o motor em funcionamento e começa a mover-se com ele, sem precisar saber como funciona todo o processo mecânico. A transferência do movimento dos eixos para as rodas, a sincronização das explosões dos pistões, a injeção de combustível na câmara de combustão, tudo fica a cargo do motor. Um motor de game (*Game Engine*) é mais ou menos isso... Dentro do conceito de engenharia de software, é a parte do projeto que executa certas funcionalidades para um programa. Dentro da área de jogos, um motor se encarregará por entender-se com o hardware gráfico, irá mandar os modelos para serem visualizados, cuidará da entrada de dados do jogador,

tratará de toda a matemática "suja" e outras coisas que o desenvolvedor de jogos normalmente não deseja fazer ou não tem tempo para se preocupar.

Existem inúmeras definições para um motor. Em todos os casos, essas definições convergem em algumas características:

- Permitir que o desenvolvedor possa criar diversos jogos diferentes, usando um mesmo motor. É comum, entretanto, que os motores sejam catalogados de acordo com os tipos de jogos para os quais eles foram concebidos [8].
- Poder reaproveitar com facilidade o código desenvolvido para um jogo.
- Abstrair a manipulação de APIs (embora, em muitos casos, o desenvolvedor use as próprias APIs dentro do ambiente do motor para implementar funcionalidades específicas, até então ainda não disponibilizadas).
- Possibilitar uma fácil integração entre código e modelagem 3D. Para esta finalidade é comum que os motores apresentem editores de fases.

Desenvolver um motor é uma tarefa complexa e cheia de desafios. Normalmente, um motor possui diversos componentes, cada um responsável por alguma etapa do processo de criação de um jogo. Os componentes mais comuns [19] de se encontrar em motores são os seguintes:

- Núcleo do Motor: consiste no "coração do motor". Este será um programa que executará a aplicação do jogo, manipulará a fase e os objetos, realizará a computação gráfica para a cena etc. Fazendo uma analogia grosseira, pode-se dizer que o núcleo do motor é o sistema operacional do jogo.
- SDK do motor: é o código-fonte do motor. Através dele pode-se alterar o funcionamento ou acrescentar novos recursos. Normalmente este componente é o mais protegido e, para consegui-lo, no caso de motores comerciais, será necessário normalmente comprar o pacote mais caro que a empresa oferece. É possível criar jogos sem o SDK de um motor, entretanto, os tipos de aplicações possíveis de serem desenvolvidos serão muito mais restritos.
- Editores de fase: através deste componente será possível unificar modelagens feitas em diversos programas, associá-los à programação, inserir códigos em scripts etc. Em muitos casos, dentro desses editores é possível também criar modelos 3D.

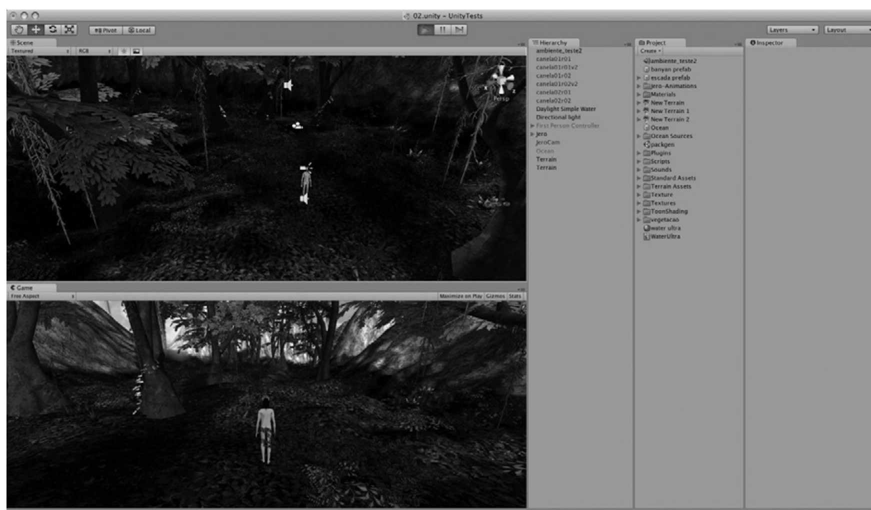


Figura A.7 Exemplo de um editor de fases (extraído do motor Unity 3D).

- Conversores / Exportadores: os *assets* serão normalmente feitos em diversos softwares comerciais, como apresentado anteriormente. Mais ainda: numa equipe de desenvolvimento grande, cada artista poderá criar os elementos no programa de sua preferência. Assim sendo, os motores deverão fornecer instrumentos para importar esses modelos para o seu formato específico. Os conversores poderão ser *plugins* instalados nos programas de modelagem 3D ou podem estar incluídos no editor de fases.
- *Builders*: Nos jogos, de forma a otimizar o processamento do jogo, serão feitos alguns pré-processamentos sobre os objetos (tais como o BSP, *lightmaps*, portais, PVS etc.). Dessa maneira, um motor fornecerá as ferramentas para realizar estes pré-processamentos. É comum que esses recursos estejam dentro do editor de fases.
- Linguagens Script: Grande parte do desenvolvimento da lógica do jogo e da inteligência artificial de elementos dinâmicos será implementada através de scripts e não diretamente sobre o núcleo do motor. Assim, cada motor possuirá sua linguagem de programação script, sendo comum usar linguagens correntes, tais como javascript, Python e LUA.

Um motor na realidade é composto por diversos módulos, sendo cada um responsável por tratar um tipo de problema envolvido em jogos.

Módulo de Visualização

Este módulo será responsável pelo *pipeline* gráfico, que é o processo de gerar imagens 2D partindo de modelos 3D (Figura A.8). Este processo é dividido em diversos estágios [11], sendo os mais importantes:

- Transformações 3D: neste estágio aplica-se o movimento aos modelos 3D. Este movimento consiste no seguinte: a cada passo do jogo uma matriz irá acumulando o resultado de todos os movimentos que o objeto sofreu ao longo de seu histórico. Antes de visualizar a cena, será aplicada a matriz sobre cada vértice que o compõe, posicionando-o no local correspondente naquele determinado instante.
- Projeção 3D para 2D: os vértices que compõem o objeto são coordenadas 3D, porém a imagem do modelo deverá ser desenhada numa superfície bidimensional, que é a tela do computador. Nesta etapa, os vértices do modelo serão projetados sobre o plano de projeção da câmera. É comum encontrar este estágio do *pipeline* junto com a etapa de transformações 3D, pois em última instância realizar essa projeção consiste numa aplicação de matriz de transformação também.

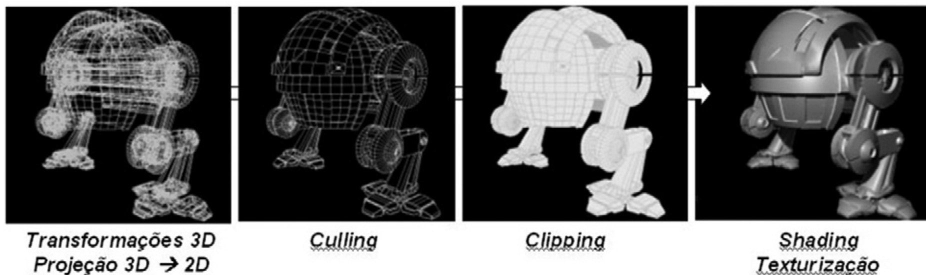


Figura A.8 Etapas mais importantes do *pipeline* gráfico.

- *Culling*: existem inúmeras formas de otimizar o processamento gráfico de um jogo. Os métodos de *culling* são algumas das abordagens possíveis (*Cull* em inglês significa "refugo, escolher, selecionar de dentro de um grupo"). Assim, o que as técnicas de *culling* terão de fazer é saber escolher polígonos adequadamente, de forma que numa determinada situação estejam presentes apenas aqueles objetos ou polígonos que realmente importam para a visualização, a partir do ponto em que a câmera se encontra. Pode-se pensar também da seguinte forma: quais dos polígonos de uma cena deverão ser enviados para o *pipeline* da placa gráfica a fim de produzir um *frame* específico? Obviamente não se deseja enviar algum

elemento que não terá nenhuma influência na visualização, mas não é tão simples realizar essa escolha de forma rápida. Existem muitos algoritmos para essa tarefa, sendo um dos mais importantes o algoritmo de *Octree*. Esse método consiste em dividir a cena em células menores. Cada uma delas pode, por sua vez, sofrer uma nova subdivisão, até que se atinjam regiões onde não há um número muito grande de objetos ou polígonos. Durante a execução do motor, em vez de mandar todos os objetos da cena para o cálculo de visualização, buscam-se apenas as células que estejam dentro do campo de visão da câmera, enviando-se em seguida apenas os objetos que estão dentro dessas células. Em muitos casos, a eficiência desse procedimento estará atrelada ao tipo de agrupamento e ordem de polígonos (um terreno possui uma distribuição de polígonos completamente diferente de um personagem ou um labirinto, por exemplo). O *culling* pode ser feito em qualquer estágio do *pipeline* gráfico. Entretanto, pode-se pensar que quanto antes se conseguir eliminar polígonos que não interessam, melhor (ninguém gosta de andar com um elefante branco nas costas de graça...). Vale a pena ressaltar que um método de *culling* não anula outro: podem-se ter os efeitos somados em muitos casos.

– Clipping: ao projetar polígonos sobre o plano de projeção da câmera, alguns polígonos cairão totalmente dentro da área da tela, e outros cairão parcialmente dentro, ou seja, apenas uma parte do polígono estará na tela de projeção. Para esses polígonos é necessário realizar o clipping (recorte), que consiste em criar novas arestas e vértices.
– Rasterização (iluminação e texturização): finalmente, a última etapa do processo de renderização consiste em preencher os polígonos adequadamente, aplicando o material com o qual estão definidos. Inicialmente poderíamos pensar em fazer esse processo através de um cálculo de iluminação para cada um dos pixels (*per pixel*) do interior de um polígono. Entretanto isso seria demasiadamente caro em termos computacionais. Para viabilizar o processo realiza-se uma interpolação (rasterização) entre a cor de cada um dos vértices que compõem o polígono. Dessa forma, o cálculo de iluminação é feito apenas para cada vértice (*per vertex*) visível da malha. Essa rasterização também poderia demandar bastante tempo de processamento, pois apesar de ser algo simples de ser feito, existem muitos pixels numa tela. O processo é realizado por um hardware específico para essa tarefa, que é a placa gráfica, ou GPU – *Graphic Processor Unit*.

Existem uma série de efeitos de iluminação (*blur*, *bump-mapping*, reflexo etc...) que não podem ser aplicados realizando-se uma rasterização como foi mencionado anteriormente. Isso porque para tais efeitos é necessário realizar um cálculo de iluminação para cada pixel e não para cada vértice. Para solucionar esse tipo de problema, as placas gráficas mais modernas possuem a capacidade de suportar *pixel shaders,* que são pequenos programas executados para cada pixel que será plotado na tela. Para mais informações sobre essa tecnologia, ver [18].

Módulo de Física

Grande parte da interatividade de um jogo se deve ao funcionamento das leis da física (ou ao menos a uma parte delas...) sobre o mundo virtual criado. Assim, ao andar sobre um labirinto e bater numa parede o jogador não pode atravessá-la; ao dar um pulo, o jogador deve colidir com o chão e não continuar caindo para sempre; ao acelerar um carro, sua velocidade deverá ir crescendo gradualmente e não abruptamente.

Os cálculos de física básicos num jogo são:

- Colisão: objetos 3D devem colidir com outros. Esta colisão não é tão trivial de ser realizada para um mundo virtual como o é para o mundo real, já que em última instância corresponderia em verificar se cada um dos polígonos de um determinado objeto possui interseção com cada um dos polígonos do restante da cena. Existem inúmeras formas de otimizar os cálculos (ver [9]), sendo a mais comum a técnica de *bounding-boxes*, que consiste em encapsular cada objeto por uma caixa e calcular a colisão para a caixa e não para a malha completa do objeto.
- Resultante de forças: os objetos se movimentam num mundo real devido à aplicação de diversas forças sobre eles. Num ambiente virtual será necessário simular a aplicação de forças de diversas naturezas sobre os objetos, calculando a resultante a cada instante a fim de verificar como será o seu movimento. Para mais detalhes, ver [3].

Os cálculos de física exigem normalmente um grande poder computacional. Assim, nas arquiteturas de motores mais modernos, está se tornando comum utilizar a GPU para realizar esses cálculos. Tal opção é interessante, devido ao alto grau de paralelismo que as arquiteturas possuem.

Módulo de Áudio

Este componente do motor permitirá o controle sobre os arquivos de som da biblioteca de *assets* do jogo. Normalmente será utilizada alguma API adequada para manipular esse tipo de arquivos, tais como o *DirectSound* ou o *OpenAL*. Além de permitir abrir e tocar os arquivos, os módulos de áudio oferecem um controle de som posicional, permitindo que objetos da cena emitam sons e estes se comportem conforme o posicionamento do objeto na cena.

Módulo de Inteligência Artificial

Entende-se por Inteligência artificial para jogos os programas que controlarão o comportamento de entidades não controladas pelo jogador, tipicamente os NPC (*non-player characters*).

Na maioria dos casos, essa inteligência artificial é desenvolvida através de máquinas de estados. Os algoritmos de máquinas de estados procuram resolver problemas formalizando diversos possíveis estados em que um elemento pode se encontrar (no caso de uma televisão, por exemplo, podem-se ter estes estados: desligada, acesa e *stand-by*). A transição de um estado para outro estará atrelado a algum evento que dispare a mudança (no exemplo anterior, ao apertar a tecla <on> a TV muda do estado de desligada para *Stand-by*).

Dessa maneira, a inteligência de um personagem de um jogo pode ser descrita por diversos estados em que ele pode se encontrar (no caso de um jogo de ação, poderíamos descrever estados como espera, perseguição, ataque, fuga, morte, por exemplo). O personagem deverá fazer um monitoramento constante sobre acontecimentos que possam disparar a mudança de um estado para outro. Usando o exemplo de um jogo de ação, suponha-se que o fato de o jogador aproximar-se mais do que 10 metros do NPC faça com que o mesmo passe do estado de espera para o estado de perseguição, tal como ilustrado na Figura A.9.

Outro caso de inteligência artificial tipicamente presente em jogos são os algoritmos de *Pathfinding,* também conhecidos como algoritmos de melhor caminho. Os NPCs, para mover-se de um lugar a outro, precisam saber como chegar até o seu objetivo. Para isso, deverão planejar e traçar um percurso, capaz de evitar os obstáculos e tentar chegar com o menor custo possível.

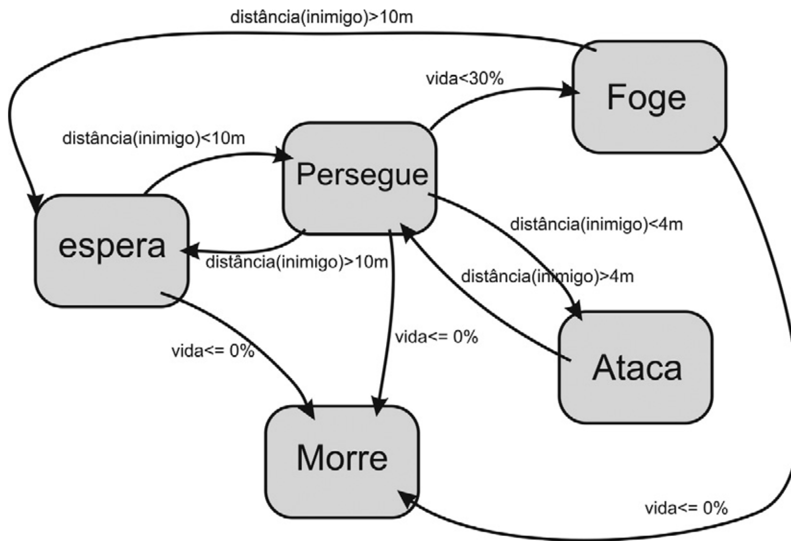


Figura A.9 Exemplo de uma máquina de estados típica para um jogo de ação.

A.5. Motores de jogos: Qual usar?

Existe uma grande quantidade de motores de jogos. Em [6] há uma excelente lista de referências sobre os principais e mais completos. Mas como escolher o motor adequado para desenvolver um jogo? Existem diversos fatores a serem considerados:

- Orçamento disponível: existem motores que custam alguns poucos dólares e outros que ultrapassam a cifra de 1 milhão. Os caros, além de possuírem todo tipo de recurso que um motor é capaz de ter, normalmente são programas que possuem uma excelente equipe de apoio, que irá acompanhar a empresa desenvolvedora do início ao fim da sua produção. Esse acompanhamento consistirá em desenvolver *plugins* específicos, adaptar funcionalidades do motor para as necessidades específicas e até dar treinamento para os programadores.
- Tipo de jogo a ser desenvolvido: apesar de existirem motores que são capazes de construir tipos bastante diferente de jogos, a maioria terá uma série de características que favorecem para o desenvolvimento de um tipo específico de jogo.
- *Milestones*: o tempo que se possui para produzir um jogo influencia muito na escolha do motor. Alguns deles permitem que um jogo seja feito em poucos dias, mas implicará numa série de soluções padrões,

que o tornam semelhante a muitos outros jogos produzidos com aquele mesmo motor.

- Plataforma: um jogo pode ser desenvolvido para diversas plataformas, tais como PC, XBOX 360, Nintendo Wii, Playstation 3, PSP, DS etc. É fundamental que o motor escolhido suporte a plataforma para a qual se está desenvolvendo. Os motores mais caros normalmente são capazes de produzir jogos para diversas plataformas.
- Documentação oferecida: o desenvolvedor de jogos deverá, antes de adotar um motor, verificar como é sua documentação. Sem uma documentação eficiente, dificilmente o programador será capaz de utilizar adequadamente os recursos oferecidos pela ferramenta.
- Ferramentas disponíveis: cada motor possuirá conversores e exportadores, que permitirão que sejam utilizados outros programas para programar seu SDK ou para modelar os objetos 3D. É fundamental que o desenvolvedor analise quais as ferramentas que podem ser usadas para o motor em questão (se a equipe de modeladores conhece apenas o MAYA, é conveniente que o motor a ser adotado suporte o MAYA para os modelos 3D).

Bibliografia

www.3dgamestudio.com

www.aliaswavefront.com

Bergen, Gino Van Den. *Collision Detection in Interactive 3D Environments* (Morgan Kaufmann Series in Interactive 3D Technology). Morgan Kaufmann. Outubro de 2003.

unity3d.com

Crawford, Chris. *The Art Of Computer Game Design*. Kindle Edition, 2009.

www.devmasters.net/engine

www.discreet.com

Eberly David H. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. 2. ed. Morgan Kaufmann, September, 2006.

Millington, Ian. "Game Physics Engine Development." *The Morgan Kaufmann Series in Interactive 3D Technology*. Março deh, 2007.

www.e-onsoftware.com

LaMothe, André. *Tricks of the 3D Game Programming Gurus-Advanced 3D Graphics and Rasterization*. Pearson Education, Junho de 2003.

www.blender.org

www.garagegames.com/

www.planetside.co.uk/terragen/

Rollings, Andrew e Morris, Dave. *Game Architecture and Design: A New Edition*. New Riders Publishers, 2004.

Schwab, Brian. *AI Game Engine Programming* (Game Development Series). Charles River Media, Setembro, 2004.

www.softimage.com/xsi

St-Laurent, Sebastien. *Shaders for Game Programmers and Artists*. Muska & Lipman/Premier-Trade; 1· ed. Maio, 2004.

Zerbst, Stefan & Düvel, Oliver. *3D Game Engine Programming*. Thomson Course Technology, Premier press, 2004.

Huizinga, Joahn. *Homo Ludens – O jogo como elemento da cultura*. Ed. Perspectiva, 5. ed., 2001.

Koster, Raph. *Theory of Fun for Game Design*. Paraglyph Press, 2004.

B
Sugestões de Projetos

B.1. Frogger

Frogger é um jogo criado no início da década de 1980, numa parceria entre as recém-criadas empresas de informática SEGA e Konami. Inicialmente foi fabricado para máquinas de arcade, porém se tornou popular quando foi portado para o Atari 2600, tornando-se também um de seus jogos mais vendidos. A mecânica do jogo é simples: o jogador deve controlar um sapo que precisa atravessar uma movimentada avenida. Obviamente o sapo não tem a vida toda, e um relógio indica o tempo que lhe resta para realizar a travessia. Nessa avenida existem diversos tipos de veículos: carros, motos e caminhões, sendo que em fila da rua os carros andam numa velocidade diferente. Cada vez que o sapo consegue atravessar a rua, ele passa para a fase seguinte, em que os veículos sofrem um ligeiro acréscimo na sua velocidade.

Desenvolva uma versão do Frogger, levando em consideração o objetivo descrito e as seguintes características:

1. O sapo deve ser um *sprite* animado, mas a sua animação somente deve ser executada quando ele está em movimento.
2. Estipule um número de vidas, que deverá ser mostrado num contador na tela. O sapo terá apenas esse número de chances para poder atravessar a rua.

3. Sempre que o sapo for atropelado, deve-se colocá-lo na posição original novamente (começo da rua).
4. Quando sua versão estiver funcionando, acrescente a seguinte funcionalidade extra: implemente um carro que inesperadamente troca de faixa, causando um elemento surpresa ao jogador.



Figura B.1 Tela do jogo *Frogger* (extraído do Atari 2600).

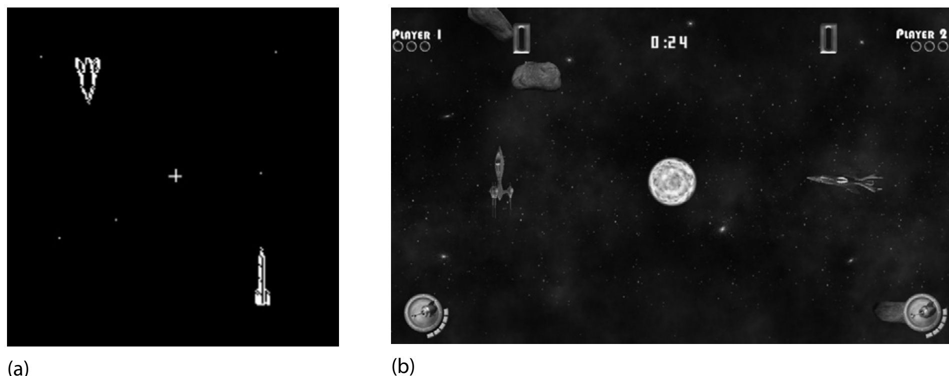
B.2. SpaceWar!

O Spacewar! é um dos jogos mais antigos da história do video game, tendo, no entanto, centenas de implementações feitas, inclusive nos consoles atuais. O jogo foi criado por Steve Russell, no ano de 1961, no MIT, usando um DEC PDP-1, um dos primeiros computadores da história (essa máquina tinha 4Kb de memória e uma velocidade de processamento de 200KHz...). O jogo consiste numa batalha de duas naves, cada uma podendo atirar na outra e também mover-se e rotacionar livremente. No centro da tela está o Sol, cuja gravidade atrai as naves. Se uma delas se chocar com a estrela, o jogador perde. Cada nave tem um número limitado de tiros e combustível.

Desenvolva uma versão do Spacewar! levando em consideração os objetivos descritos e as seguintes características:

1. Para um inimigo destruir o outro é necessário mais do que um só tiro.
2. Crie alguns elementos extras ao jogo, tais como meteoros e cometas que podem danificar a nave.

3. Quando o jogo estiver funcionando, crie uma segunda versão, sendo uma das naves controlada pelo computador.
4. Nessa nova versão, acrescente às naves uma inércia no movimento, de forma que mesmo que o jogador pare de mover a nave, ela continue se movendo por alguns instantes.



(a) (b)

Figura B.2 (a) Primeira versão do Spacewar! numa tela do DEP PDP-1; (b) versão mais recente do jogo (extraído do Microsoft XNA).

B.3. Tetris

O Tetris é um dos jogos mais populares que existem, já tendo vendido mais de 100 milhões de cópias para mais de 50 plataformas diferentes. Este jogo foi inventado pelos russos Alexey Pajitnov, Dmitry Pavlovsky e Vadim Gerasimov, sendo este último um estudante de computação de 16 anos. Apesar de ser um jogo simples, o desafio que impõe ao jogador lhe dá uma excelente jogabilidade.

O objetivo do jogo é encaixar os chamados *tetraminós*, que são peças de diversos formatos e cores que descem do topo da tela. Quando uma linha é completada, esta desaparece e são computados pontos para o jogador. O jogador perde quando as linhas incompletas se empilham até o topo da tela do jogo.

Desenvolva uma versão do Tétris levando em consideração o objetivo descrito e as seguintes características:

1. Crie um conjunto de tipos de peças. Pode-se criar uma estrutura que diga como é cada peça, em termos geométricos. É importante que todas elas sejam combinações de blocos quadrados. Nas primeiras implementações, você poderá criar um conjunto pequeno de peças, que posteriormente podem ser incrementadas. Cada tipo pode ter associado a si um número inteiro.

2. Uma forma de mapear as peças no tabuleiro consiste em criar uma matriz de números inteiros, que no começo pode ser inicializada com zero. À medida que uma peça "estaciona" num local, esta matriz será atualizada.
3. A cada frame, o jogo deverá percorrer a matriz e verificar se uma linha de peças ficou completa. Seguindo a lógica proposta no item 2, este passo corresponde a verificar se uma linha ainda possui zeros ou não.
4. O método draw() do *Game Controller* pode usar esta matriz para saber o que desenhar na tela, a cada frame.
5. O método step() do *Game Controller* pode ser usado para fazer com que todas as peças caiam um nível. Tome cuidado para manter um tempo constante em cada passo de queda.

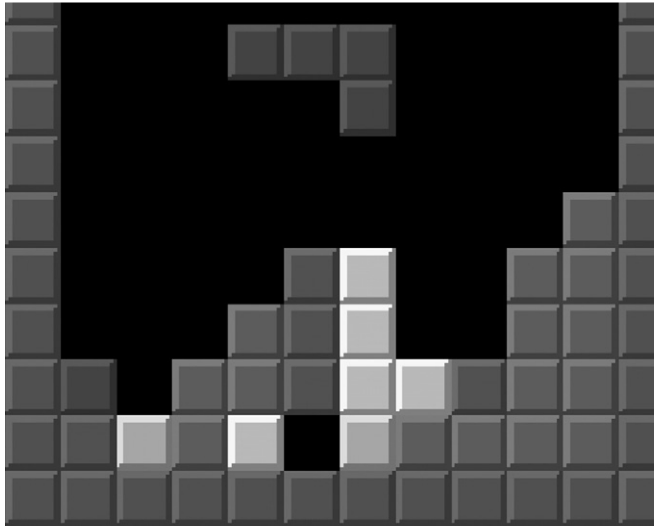


Figura B.3 Tela do jogo Tétris. (extraído da *Wikipedia*).

C

O Motor 2D javaPlay

O motor 2D javaPlay foi especialmente desenvolvido para este livro e tem a estrutura de um motor profissional. O completo entendimento do seu funcionamento requer algum tempo de estudo. A ideia é que você comece usando as suas funcionalidades mais gerais, sem precisar entrar em detalhes. Depois você vai esmiuçando cada uma das classes, com auxílio da Parte III do livro (página Web). Para um estudo mais detalhado, veja o Apêndice D, que contém os princípios de projeto do javaPlay.

Neste apêndice você encontra o código e as instruções das versões mais básicas que servem de ponto de partida para projetos mais completos.

Todas as versões são compostas por dois pacotes:

- Demo
- javaPlay

Para criar um jogo, você deve alterar o **Demo**. As primeiras alterações podem ser inclusões de Game Objects. O **Demo** já vem com um GameObject chamado Player. No método step de SimpleController.java está a movimentação do Player através do teclado para cada tick do jogo. Existem três versões do Motor, da menos completa à mais completa: Versão Mini, Versão 00 e Versão 0. Apenas na Versão 0, a mais completa, o

método step de SimpleController.java (pacote Demo) tem tratamento da colisão.

O módulo **javaPlay** é o motor propriamente dito. Na versão Mini, o javaPlay é bastante reduzido (para fins didáticos). Nas outras versões o javaPlay é completo e não muda.

C.1. JavaPlay — Versão Mini

Esta é a versão mais simples do Motor (Figura C1), que basicamente implementa um loop. O módulo DemoMini contém apenas as classes Main.java e Player.java. No Player.java temos os seguintes métodos: Player, step e draw. No método `Player()` definimos x e y iniciais (e criamos e inicializamos outras variáveis que queremos usar). No método draw, desenhamos o que queremos que o GameObject Player seja (no caso: pintamos o canvas de branco, desenhamos o player como um quadrado preto na coordenada (x,y), desenhamos dois retângulos como se fossem paredes e desenhamos um círculo na coordenada (x2,y2). No método step, definimos o que deve mudar a cada tick do jogo (no caso, as coordenadas (x,y) e (x2,y2)). O javaPlay contém as versões mais básicas das três classes indispensáveis: GameCanvas.java, GameEngine.java e GameObject.java. Também estão indicadas as modificações que poderiam ser feitas para usar sprites, dando a esta versão Mini um aspecto de um *Game Engine* de fato. Note que o exercício de desenhar o player como um quadrado preto é apenas didático, visto que os únicos desenhos em um verdadeiro motor de jogos 2D são carregamentos de imagens (background, tiles e sprites).

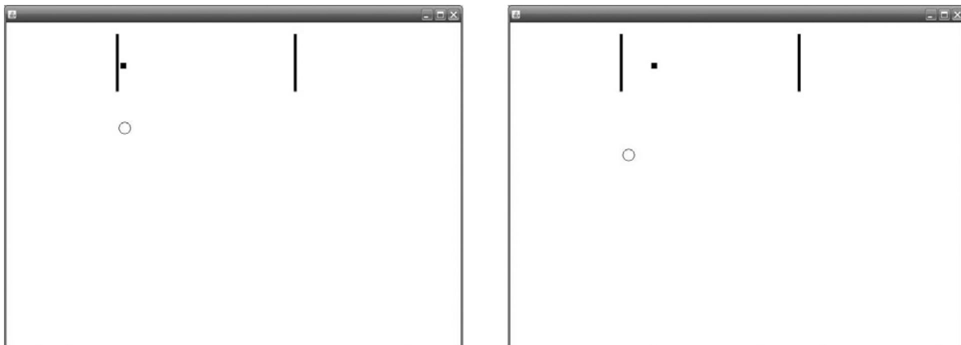


Figura C1 Janelas do demo do motor Nível Mini.

C.1.1. Demo da Versão Mini

Main.java

```
/*
 * Main
 */

package demo;

import javaPlay.GameEngine;

public class Main
{
    public static void main(String[] args)
    {
        Player player = new Player();

        GameEngine.getInstance().addGameObject(player);

        GameEngine.getInstance().run();
    }
}
```

Player.java

```
/*
 *  GameObject: player
 */

package demo;

import java.awt.Color;
import java.awt.Graphics;
// import java.util.logging.Level;   // if Sprite is required
// import java.util.logging.Logger;  // if Sprite is required
import javaPlay.GameEngine;
import javaPlay.GameObject;
// import javaPlay.Sprite;

/*
 * @author VisionLab/PUC-Rio
 */
public class Player extends GameObject
{
//    private Sprite sprite;
    private int delta;
    private int min;
    private int max;
```

```
    private int x2, y2; // circulo extra

    public Player()
    {
/* Descomente para sprites
        try
        {
            sprite = new Sprite("player.png", 8, 32, 32);
        }
        catch (Exception ex)
        {
          Logger.getLogger(Player.class.getName()).log(Level.SEVERE, null, ex);
        }
*/
        x = 200;
        y = 100;

        x2 = 200;  // circulo extra
        y2 = 200;  // circulo extra

        delta = 1;
        min = x;
//        max = GameEngine.getInstance().getGameCanvas().getWidth();  // =
          canvas width
        max = 500;
    }

    public void step(long timeElapsed)
    {
        x += delta;
        y2 += delta;
        if(x > max)
            delta *= -1;
        if(x < min)
            delta *= -1;

    }

    public void draw(Graphics g)
    {
        // canvas in white
        g.setColor(Color.WHITE);
        g.fillRect(0, 0, GameEngine.getInstance().getGameCanvas().getWidth(),
                GameEngine.getInstance().getGameCanvas().getHeight());
        // draw player as a back square
        g.setColor(Color.BLACK);
        g.fillRect(x, y, 10, 10);
        // draw walls
```

```
        g.fillRect(195, 50, 5, 100);
        g.fillRect(505, 50, 5, 100);
        // draw a circle
        g.drawOval(x2,y2,20,20);

//        sprite.draw(g, x, y);
    }
}
```

C.1.2. javaPlay reduzido para a Versão Mini

GameCanvas.java

```
/*
 * GameCanvas
 */

package javaPlay;

import java.awt.Canvas;
import java.awt.Container;
import java.awt.Graphics;
import java.awt.GraphicsConfiguration;
import java.awt.GraphicsDevice;
import java.awt.GraphicsEnvironment;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.Toolkit;
import java.awt.image.BufferStrategy;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JFrame;

/**
 * @author VisionLab/PUC-Rio
 */
public class GameCanvas extends JFrame
{
    private final int defaultScreenWidth = 800;
    private final int defaultScreenHeight = 600;
    private Graphics g;
    private BufferStrategy bf;
    private int renderScreenStartX;
    private int renderScreenStartY;

    public GameCanvas(GraphicsConfiguration gc)
    {
        super(gc);
```

```
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(defaultScreenWidth, defaultScreenHeight);
        setVisible(true);

        createBufferStrategy(2);

        renderScreenStartX = this.getContentPane().getLocationOnScreen().x;
        renderScreenStartY = this.getContentPane().getLocationOnScreen().y;

        bf = getBufferStrategy();
    }

    public int getRenderScreenStartX()
    {
        return renderScreenStartX;
    }

    public int getRenderScreenStartY()
    {
        return renderScreenStartY;
    }

    public Graphics getGameGraphics()
    {
        g = bf.getDrawGraphics();
        return g;
    }

    public void swapBuffers()
    {
        bf.show();
        g.dispose();
        Toolkit.getDefaultToolkit().sync();
    }
}
```

GameEngine.java

```
/*
 * GameEngine
 */

package javaPlay;

import java.awt.Graphics;
import java.awt.GraphicsConfiguration;
import java.awt.GraphicsDevice;
import java.awt.GraphicsEnvironment;
import java.util.*;
```

```
/**
 * @author VisionLab/PUC-Rio
 */
public class GameEngine
{
    private GameCanvas canvas;
    private long lastTime;
    private boolean engineRunning;
    private static GameEngine instance;
    private Vector gameObjects;

    private GameEngine()
    {
        GraphicsEnvironment graphEnv = GraphicsEnvironment.
        getLocalGraphicsEnvironment();
        GraphicsDevice graphDevice = graphEnv.getDefaultScreenDevice();
        GraphicsConfiguration graphicConf = graphDevice.
        getDefaultConfiguration();

        canvas = new GameCanvas(graphicConf);

        lastTime = System.currentTimeMillis();
        engineRunning = true;
        instance = null;

        gameObjects = new Vector();
    }

    public static GameEngine getInstance()
    {
        if(instance == null)
        {
            instance = new GameEngine();
        }
        return instance;
    }

    public GameCanvas getGameCanvas()
    {
        return canvas;
    }

    public void requestShutdown()
    {
        engineRunning = false;
    }

    public void addGameObject(GameObject object)
```

```
    {
        gameObjects.add(object);
    }

    public void removeGameObject(GameObject object)
    {
        gameObjects.remove(object);
    }

    public void run()
    {
        long currentTime;

        while(engineRunning == true)
        {
            currentTime = System.currentTimeMillis();

            for(int i = 0 ; i < gameObjects.size() ; i++)
            {
                GameObject obj = (GameObject)gameObjects.elementAt(i);

                obj.step(currentTime - lastTime);

                obj.draw(canvas.getGameGraphics());
            }

            canvas.swapBuffers();
        }

        canvas.dispose();
    }
}
```

GameObject.java

```
/*
 * GameObject
 */

package javaPlay;

import java.awt.Graphics;

/**
 * @author VisionLab/PUC-Rio
 */
public abstract class GameObject
{
```

```
    public int x;
    public int y;

    public abstract void step(long timeElapsed);
    public abstract void draw(Graphics g);
}
```

C.2. Demo Versão 00

A Versão 00 tem apenas um jogador (Player) e não trata colisão (Figura C2).

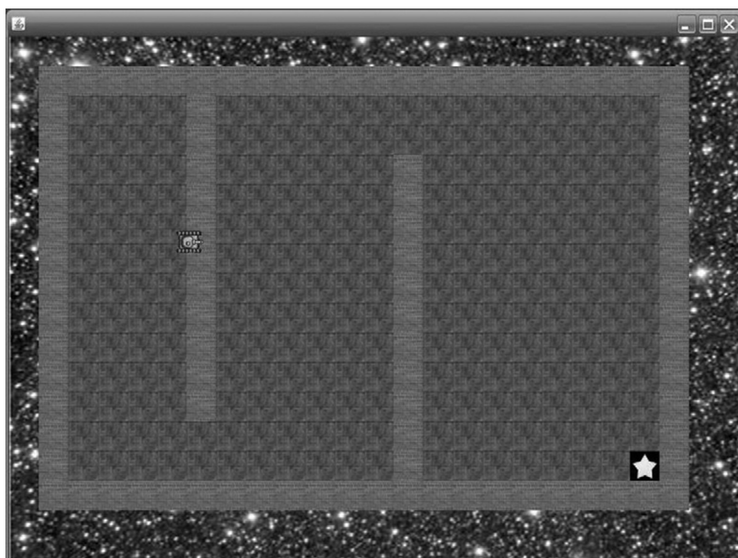


Figura C2 Janela do demo do motor Versão 00.

O jogo se resume a passear com o jogador (*Player*) em um cenário de tiles, sem colisões. Os elementos gráficos são os seguintes:

1. imagem do jogador (player.png de tamanho 256x32)
2. imagem do fundo (backdrop.jpg de tamanho 800x600)
3. imagens de Tiles:
   muro: wall.jpg de tamanho 32x32
   chão: floor.jpg de tamanho 32x32
   alvo (em forma de estrela): end.jpg de tamanho 32x32

O cenário é definido através de um arquivo texto (com nome scene.scn) no seguinte formato:

número-de-tiles
imagem-do-tile-1
...
imagem-do-tile-n
mapa-de-localização
%
imagem-do-fundo

No exemplo da FiguraC2, o arquivo scene.scn é o seguinte:

```
3
wall.jpg
floor.jpg
end.jpg
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
0,1,2,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1
0,1,2,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1
0,1,2,2,2,2,1,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,1
0,1,2,2,2,2,1,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,1
0,1,2,2,2,2,1,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,1
0,1,2,2,2,2,1,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,1
0,1,2,2,2,2,1,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,1
0,1,2,2,2,2,1,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,1
0,1,2,2,2,2,1,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,1
0,1,2,2,2,2,1,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,1
0,1,2,2,2,2,1,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,1
0,1,2,2,2,2,2,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,1
0,1,2,2,2,2,2,2,2,2,2,2,2,1,2,2,2,2,2,2,2,3,1
0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
%
backdrop.jpg
```

Você deve notar o *tile* número 3 (o alvo end.jpg) em negrito no canto inferior direito.

Player.java

```
/*
 * Game Object: Player
 */

package demo;

import java.awt.Graphics;
import javaPlay.GameObject;
import javaPlay.Sprite;

/**
 * @author VisionLab/PUC-Rio
 */
public class Player extends GameObject
{
    private Sprite sprite;
    private int currFrame;

    private final int UP_FRAMES = 0;
    private final int LEFT_FRAMES = 2;
    private final int RIGHT_FRAMES = 6;
    private final int DOWN_FRAMES = 4;

    public void setSprite(Sprite sprite)
    {
        this.sprite = sprite;
        currFrame = 0;
    }

    public void moveUp()
    {
        y--;
        currFrame = UP_FRAMES;
    }

    public void moveDown()
    {
        y++;
        currFrame = DOWN_FRAMES;
    }

    public void moveLeft()
    {
        x--;
        currFrame = LEFT_FRAMES;
    }
```

```
    public void moveRight()
    {
        x++;
        currFrame = RIGHT_FRAMES;
    }

    public void step(long timeElapsed)
    {

    }

    public void draw(Graphics g)
    {
        sprite.setCurrAnimFrame(currFrame);
        sprite.draw(g, x, y);
    }
}
```

SimpleController

```
/*
 * SimpleController
 */

package demo;

import java.awt.Graphics;
import java.util.logging.Level;
import java.util.logging.Logger;
import javaPlay.GameEngine;
import javaPlay.GameStateController;
import javaPlay.Keyboard;
import javaPlay.Scene;
import javaPlay.Sprite;

/**
 *
 * @author VisionLab/PUC-Rio
 */
public class SimpleController implements GameStateController
{
    private Sprite playerSprite;
    private Player player;
    private Scene scene;

    public void load()
    {
        try
```

```
        {
            playerSprite = new Sprite("tankplayer.png", 8, 32, 32);

            scene = new Scene();
            scene.loadFromFile("scene.scn");

            player = new Player();
            player.setSprite(playerSprite);

            scene.addOverlay(player);
        }
        catch (Exception ex)
        {
            Logger.getLogger(SimpleController.class.getName()).log(Level.
            SEVERE, null, ex);
        }
    }

    public void unload()
    {

    }

    public void start()
    {
        player.x = 92;
        player.y = 114;
    }

    public void step(long timeElapsed)
    {
        Keyboard k = GameEngine.getInstance().getKeyboard();

        if(k.keyDown(Keyboard.UP_KEY) == true)
        {
            player.moveUp();
        }
        if(k.keyDown(Keyboard.DOWN_KEY) == true)
        {
            player.moveDown();
        }
        if(k.keyDown(Keyboard.LEFT_KEY) == true)
        {
            player.moveLeft();
        }
        if(k.keyDown(Keyboard.RIGHT_KEY) == true)
        {
            player.moveRight();
        }
```

```
        player.step(timeElapsed);
    }

    public void draw(Graphics g)
    {
        scene.draw(g);
    }

    public void stop()
    {

    }
}
```

EndScreenControl.java

```
/*
 * EndScreenController
 */

package demo;

import java.awt.Graphics;
import java.util.logging.Level;
import java.util.logging.Logger;
import javaPlay.GameEngine;
import javaPlay.GameStateController;
import javaPlay.Keyboard;
import javaPlay.Sprite;

/**
 * @author VisionLab/PUC-Rio
 */
public class EndScreenController implements GameStateController
{
    private Sprite endScreen;

    public void load()
    {
        try
        {
            endScreen = new Sprite("finalback.jpg", 1, 800, 600);
        }
        catch (Exception ex)
        {
            Logger.getLogger(EndScreenController.class.getName()).log(Level.
            SEVERE, null, ex);
        }
    }
```

```
    public void unload()
    {

    }

    public void start()
    {

    }

    public void step(long timeElapsed)
    {
        Keyboard k = GameEngine.getInstance().getKeyboard();

        if(k.keyDown(Keyboard.ENTER_KEY) == true)
        {
            GameEngine.getInstance().setNextGameStateController
            (Global.START_MENU_CONTROLLER_ID);
        }
    }

    public void draw(Graphics g)
    {
        endScreen.draw(g, 0, 0);
    }

    public void stop()
    {

    }
}
```

Global.java

```
/*
 *  Global
 */

package demo;

/**
 * @author VisionLab/PUC-Rio
 */
public class Global
{
    public static final int START_MENU_CONTROLLER_ID = 0;
    public static final int SIMPLE_CONTROLLER_ID = 1;
    public static final int END_SCREEN_CONTROLLER_ID = 2;
}
```

Main.java

```
/*
 * Main
 */

package demo;

import javaPlay.GameEngine;

public class Main
{
    public static void main(String[] args)
    {
        GameEngine.getInstance().addGameStateController
        (Global.START_MENU_CONTROLLER_ID, new StartMenuController());

        GameEngine.getInstance().addGameStateController
        (Global.SIMPLE_CONTROLLER_ID, new SimpleController());

        GameEngine.getInstance().addGameStateController
        (Global.END_SCREEN_CONTROLLER_ID, new EndScreenController());

        GameEngine.getInstance().setStartingGameStateController
        (Global.START_MENU_CONTROLLER_ID);

        GameEngine.getInstance().run();
    }
}
```

StartMenuControl.java

```
/*
 * StartMenuController
 */

package demo;

import java.awt.Graphics;
import java.awt.Point;
import java.util.logging.Level;
import java.util.logging.Logger;
import javaPlay.GameEngine;
import javaPlay.GameStateController;
import javaPlay.Mouse;
import javaPlay.Sprite;

/**
 *
```

```
 * @author VisionLab/PUC-Rio
 */
public class StartMenuController implements GameStateController
{
    private Sprite menu;

    public void load()
    {
        try
        {
            menu = new Sprite("startmenu.jpg", 1, 800, 600);
        }
        catch (Exception ex)
        {
            Logger.getLogger(StartMenuController.class.getName())
            .log(Level.SEVERE, null, ex);
        }
    }

    public void unload()
    {

    }

    public void start()
    {

    }

    public void step(long timeElapsed)
    {
        Mouse m = GameEngine.getInstance().getMouse();

        if(m.isLeftButtonPressed() == true)
        {
            Point p = m.getMousePos();

            if((p.x >= 363) && (p.y >= 353) && (p.x <= 485) && (p.y <= 389))
            {
                GameEngine.getInstance().setNextGameStateController
                (Global.SIMPLE_CONTROLLER_ID);
            }
        }
    }

    public void draw(Graphics g)
    {
        menu.draw(g, 0, 0);
    }
```

```
    public void stop()
    {

    }
}
```

C.3. Demo da Versão 0

Esta versão é a mais completa, contendo um jogador com sprite animado mais elaborado, placar de vidas (Score), dois inimigos e tratamento de colisão (Figura C3). Nesta seção estão incluídas apenas as novas classes. O pacote javaPlay é o mesmo para as versões 0 e 00.

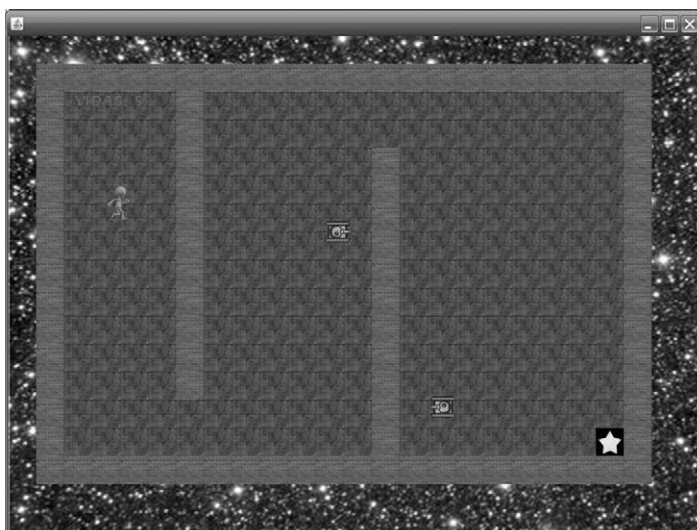


Figura C3 Janela do demo do motor Versão 0.

Neste jogo, o objetivo é chegar ao alvo em forma de estrela. A cada colisão com um inimigo, o placar geral registra perdas de vida. Quando a vida zera, o programa é fechado (shutdown). Quando o Player alcança o alvo, uma imagem de finalização é apresentada e um toque na tecla "Enter" reinicia o jogo (essas ações são consideradas na classe EndScreenController.java).

Player.java

```
/*
 * Game Object: Player
 */

package demo;
```

```
import java.awt.Graphics;
import javaPlay.GameObject;
import javaPlay.Sprite;

/**
 * @author VisionLab/PUC-Rio
 */
public class Player extends GameObject
{
    private Sprite sprite;
    private int currFrame;

    /*  animation with 8 frames - file "tankplayer.png"
    private final int[] UP_FRAMES = {0, 1};
    private final int[] LEFT_FRAMES = {2, 3};
    private final int[] RIGHT_FRAMES = {6, 7};
    private final int[] DOWN_FRAMES = {4, 5};
    private final int SHOW_STEP = 1;
    */

    /* animation with 40 frames - file "walkingplayer.png" */
    private final int[] UP_FRAMES =    { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    private final int[] LEFT_FRAMES =  {10,11,12,13,14,15,16,17,18,19};
    private final int[] RIGHT_FRAMES = {20,21,22,23,24,25,26,27,28,29};
    private final int[] DOWN_FRAMES =  {30,31,32,33,34,35,36,37,38,39};
    private final int SHOW_STEP = 30;

    private int animIndex;
    private int currMoveAnim;
    private int stepCounter;

    private final int MOVE_ANIM_UP = 0;
    private final int MOVE_ANIM_DOWN = 1;
    private final int MOVE_ANIM_LEFT = 2;
    private final int MOVE_ANIM_RIGHT = 3;

    public void setSprite(Sprite sprite)
    {
        this.sprite = sprite;
        animIndex = 0;
        currFrame = UP_FRAMES[animIndex];
        currMoveAnim = MOVE_ANIM_UP;
        stepCounter = 0;
    }

    public void moveUp()
    {
        y--;
```

```
        currMoveAnim = MOVE_ANIM_UP;
    }

    public void moveDown()
    {
        y++;
        currMoveAnim = MOVE_ANIM_DOWN;
    }

    public void moveLeft()
    {
        x--;
        currMoveAnim = MOVE_ANIM_LEFT;
    }

    public void moveRight()
    {
        x++;
        currMoveAnim = MOVE_ANIM_RIGHT;
    }

    public void step(long timeElapsed)
    {
        switch(currMoveAnim)
        {
            case MOVE_ANIM_UP:
               if (++stepCounter >= SHOW_STEP)
               {
                  animIndex = (animIndex + 1) % UP_FRAMES.length;
                  currFrame = UP_FRAMES[animIndex];
                  stepCounter = 0;
               }
               break;
            case MOVE_ANIM_DOWN:
               if (++stepCounter >= SHOW_STEP)
               {
                  animIndex = (animIndex + 1) % DOWN_FRAMES.length;
                  currFrame = DOWN_FRAMES[animIndex];
                  stepCounter = 0;
               }
               break;
            case MOVE_ANIM_LEFT:
               if (++stepCounter >= SHOW_STEP)
               {
                  animIndex = (animIndex + 1) % LEFT_FRAMES.length;
                  currFrame = LEFT_FRAMES[animIndex];
                  stepCounter = 0;
               }
```

```
                break;
            case MOVE_ANIM_RIGHT:
                if (++stepCounter >= SHOW_STEP)
                {
                    animIndex = (animIndex + 1) % RIGHT_FRAMES.length;
                    currFrame = RIGHT_FRAMES[animIndex];
                    stepCounter = 0;
                }
                break;
        }
    }
    public void draw(Graphics g)
    {
        sprite.setCurrAnimFrame(currFrame);
        sprite.draw(g, x, y);
    }
}
```

Enemy.java

```
/*
 * ObjectGame: Enemy
 */

package demo;

import java.awt.Graphics;
import java.awt.Point;
import javaPlay.GameObject;
import javaPlay.Sprite;

/**
 * @author VisionLab/PUC-Rio
 */
public class Enemy extends GameObject
{
    private Sprite sprite;
    private Point pathEndPoint1;
    private Point pathEndPoint2;
    private boolean movingToEndPoint2;
    private final int UP_FRAME = 0;
    private final int LEFT_FRAME = 2;
    private final int RIGHT_FRAME = 6;
    private final int DOWN_FRAME = 4;
    private int currFrame;

    public void setSprite(Sprite sprite)
    {
        this.sprite = sprite;
    }
```

```
public void setPath(Point endPoint1, Point endPoint2)
{
    pathEndPoint1 = endPoint1;
    pathEndPoint2 = endPoint2;
    x = pathEndPoint1.x;
    y = pathEndPoint2.y;
    movingToEndPoint2 = true;
}

public void step(long timeElapsed)
{
    Point endPoint;

    if(movingToEndPoint2 == true)
    {
        endPoint = pathEndPoint2;
    }
    else
    {
        endPoint = pathEndPoint1;
    }

    if(x < endPoint.x)
    {
        x++;
        currFrame = RIGHT_FRAME;
    }
    else if(x > endPoint.x)
    {
         x--;
         currFrame = LEFT_FRAME;
    }

    if(y < endPoint.y)
    {
        y++;
        currFrame = UP_FRAME;
    }
    else if(y > endPoint.y)
    {
        y--;
        currFrame = DOWN_FRAME;
    }

    if((x == endPoint.x) && (y == endPoint.y))
    {
        movingToEndPoint2 = !movingToEndPoint2;
    }
}
```

```
    public void draw(Graphics g)
    {
        sprite.setCurrAnimFrame(currFrame);
        sprite.draw(g, x, y);
    }
}
```

Score.java

```
/*
 * Game Object: Score
 */

package demo;

import java.awt.Graphics;
import javaPlay.GameObject;
import java.awt.Color;
import java.awt.Font;
import java.lang.String;

/**
 *
 * @author VisionLab/PUC-Rio
 */
public class Score extends GameObject
{
    private int scoreCounter = 5;

    public void setScore(int n)
    {
        scoreCounter = scoreCounter + n;
    }

    public int getScore()
    {
        return scoreCounter;
    }

    public void step(long timeElapsed)
    {

    }
    public void draw(Graphics g)
    {
        g.setColor(Color.RED);
        g.setFont(new Font("Arial", Font.BOLD, 18));
        g.drawString("VIDAS: "+String.valueOf(scoreCounter), 80, 110);
    }
}
```

SimpleController.java

```
/*
 * SimpleController
 */

package demo;

import java.awt.Graphics;
import java.awt.Point;
import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;
import javaPlay.GameEngine;
import javaPlay.GameStateController;
import javaPlay.Keyboard;
import javaPlay.Scene;
import javaPlay.Sprite;
import javaPlay.TileInfo;

/**
 *
 * @author VisionLab/PUC-Rio
 */
public class SimpleController implements GameStateController
{
    private Sprite playerSprite;
    private Sprite enemySprite;
    private Player player;
    private Enemy[] enemies;
    private Scene scene;
    private final int NUM_OF_ENEMIES = 2;
    private final int TILE_WALL = 1;
    private final int TILE_EXIT = 3;

    private int[] midx;
    private int[] midy;
    private final int UP_MID = 0;
    private final int DOWN_MID = 1;
    private final int LEFT_MID = 2;
    private final int RIGHT_MID = 3;

    private Score score;

    public void load()
    {
        try
        {
```

```
//            playerSprite = new Sprite("tankplayer.png", 8, 32, 32);
            playerSprite = new Sprite("walkingplayer.png", 40, 40, 40);

            enemySprite = new Sprite("enemy.png", 8, 32, 32);

            scene = new Scene();
            scene.loadFromFile("scene.scn");

            player = new Player();
            player.setSprite(playerSprite);

            scene.addOverlay(player);
            enemies = new Enemy[NUM_OF_ENEMIES];

            for (int i = 0; i < NUM_OF_ENEMIES; i++)
            {
                enemies[i] = new Enemy();
                enemies[i].setSprite(enemySprite);
                scene.addOverlay(enemies[i]);
            }

            score = new Score();
            scene.addOverlay(score);

            midx = new int[4];
            midy = new int[4];
        }
        catch (Exception ex)
        {
            Logger.getLogger(SimpleController.class.getName())
            .log(Level.SEVERE, null, ex);
        }
    }

    public void unload()
    {

    }

    public void start()
    {
        player.x = 92;
        player.y = 114;

        Point startPosEnemy1 = new Point();
        Point endPosEnemy1 = new Point();

        startPosEnemy1.x = 251;
        startPosEnemy1.y = 208;
```

```
        endPosEnemy1.x = 379;
        endPosEnemy1.y = 208;

        enemies[0].setPath(startPosEnemy1, endPosEnemy1);

        Point startPosEnemy2 = new Point();
        Point endPosEnemy2 = new Point();

        startPosEnemy2.x = 477;
        startPosEnemy2.y = 408;
        endPosEnemy2.x = 666;
        endPosEnemy2.y = 408;

        enemies[1].setPath(startPosEnemy2, endPosEnemy2);
    }

    public void step(long timeElapsed)
    {
        Keyboard k = GameEngine.getInstance().getKeyboard();

        if(k.keyDown(Keyboard.UP_KEY) == true)
        {
            player.moveUp();
        }
        if(k.keyDown(Keyboard.DOWN_KEY) == true)
        {
            player.moveDown();
        }
        if(k.keyDown(Keyboard.LEFT_KEY) == true)
        {
            player.moveLeft();
        }
        if(k.keyDown(Keyboard.RIGHT_KEY) == true)
        {
            player.moveRight();
        }

        player.step(timeElapsed);

        Point playerMin = new Point(player.x, player.y);
        Point playerMax = new Point(player.x + 31, player.y + 31);

        for(int i = 0 ; i < NUM_OF_ENEMIES ; i++)
        {
            enemies[i].step(timeElapsed);

            Point enemyMin = new Point(enemies[i].x, enemies[i].y);
            Point enemyMax = new Point(enemies[i].x + 31, enemies[i].y + 31);
```

```
        if(isColliding(playerMin, playerMax, enemyMin, enemyMax) == true)
        {
            // update Score
            score.setScore(-1);
            // back to start point
            player.x = 92;
            player.y = 114;
            // Shutdown if Score = 0
            if (score.getScore() == 0)
                GameEngine.getInstance().requestShutdown();

            return;
        }
    }

    Vector tiles = scene.getTilesFromRect(playerMin, playerMax);

    for(int i = 0 ; i < tiles.size() ; i++)
    {
        TileInfo tile = (TileInfo)tiles.elementAt(i);

        if((tile.id == TILE_WALL) && (isColliding(playerMin, playerMax,
                tile.min, tile.max) == true))
        {
            int centerx = playerMin.x/2 + playerMax.x/2;
            int centery = playerMin.y/2 + playerMax.y/2;

            midx[UP_MID] = tile.min.x/2 + tile.max.x/2;
            midy[UP_MID] = tile.min.y;

            midx[DOWN_MID] = tile.min.x/2 + tile.max.x/2;
            midy[DOWN_MID] = tile.max.y;

            midx[LEFT_MID] = tile.min.x;
            midy[LEFT_MID] = tile.min.y/2 + tile.max.y/2;

            midx[RIGHT_MID] = tile.max.x;
            midy[RIGHT_MID] = tile.min.y/2 + tile.max.y/2;

            int dx = midx[UP_MID] - centerx;
            int dy = midy[UP_MID] - centery;
            int nearestMid = 0;
            double nearestDist = Math.sqrt(dx*dx + dy*dy);

            for(int j = 1 ; j < 4 ; j++)
            {
                dx = midx[j] - centerx;
                dy = midy[j] - centery;
                double dist = Math.sqrt(dx*dx + dy*dy);
```

```
                if(dist < nearestDist)
                {
                    nearestDist = dist;
                    nearestMid = j;
                }
            }

            switch(nearestMid)
            {
                case UP_MID:
                    player.y = tile.min.y - 32;
                    break;
                case DOWN_MID:
                    player.y = tile.max.y;
                    break;
                case LEFT_MID:
                    player.x = tile.min.x - 32;
                    break;
                case RIGHT_MID:
                    player.x = tile.max.x;
                    break;
            }
        }
        else if(tile.id == TILE_EXIT)
        {
            GameEngine.getInstance().setNextGameStateController
                    (Global.END_SCREEN_CONTROLLER_ID);
        }
    }
}

public void draw(Graphics g)
{
    scene.draw(g);
}

public void stop()
{

}

private boolean isColliding(Point min1, Point max1, Point min2, Point max2)
{
    if(min1.x > max2.x || max1.x < min2.x)
    {
        return false;
    }
    if(min1.y > max2.y || max1.y < min2.y)
```

```
        {
            return false;
        }


        return true;
    }
}
```

C.4. javaPlay para as Versões 00 e 0

GameObject.java

```
/*
 * GameObject
 */

package javaPlay;

import java.awt.Graphics;

/**
 * @author VisionLab/PUC-Rio
 */
public abstract class GameObject
{
    public int x;
    public int y;

    public abstract void step(long timeElapsed);
    public abstract void draw(Graphics g);
}
```

GameEngine.java

```
/*
 * GameEngine
 */

package javaPlay;

import java.awt.Graphics;
import java.awt.GraphicsConfiguration;
import java.awt.GraphicsDevice;
import java.awt.GraphicsEnvironment;
import java.util.*;

/**
 * @author VisionLab/PUC-Rio
 */
```

```
public class GameEngine
{
    private GameCanvas canvas;
    private Mouse mouse;
    private Keyboard keyboard;
    private long lastTime;
    private boolean engineRunning;
    private Hashtable gameStateControllers;
    private GameStateController currGameState;
    private GameStateController nextGameState;
    private static GameEngine instance;

    private GameEngine()
    {
        GraphicsEnvironment graphEnv = GraphicsEnvironment.
        getLocalGraphicsEnvironment();
        GraphicsDevice graphDevice = graphEnv.getDefaultScreenDevice();
        GraphicsConfiguration graphicConf = graphDevice.
        getDefaultConfiguration();

        canvas = new GameCanvas(graphicConf);

        mouse = new Mouse();
        keyboard = new Keyboard();

        canvas.addMouseListener(mouse);
        canvas.addMouseMotionListener(mouse);
        canvas.addKeyListener(keyboard);

        lastTime = System.currentTimeMillis();
        engineRunning = true;
        gameStateControllers = new Hashtable();
        currGameState = null;
        nextGameState = null;
        instance = null;
    }

    public static GameEngine getInstance()
    {
        if(instance == null)
        {
            instance = new GameEngine();
        }
        return instance;
    }

    public GameCanvas getGameCanvas()
```

```
    {
        return canvas;
    }

    public Mouse getMouse()
    {
        return mouse;
    }

    public Keyboard getKeyboard()
    {
        return keyboard;
    }

    public void addGameStateController(int id, GameStateController gs)
    {
        gameStateControllers.put(id, gs);

        gs.load();
    }

    public void removeGameStateController(int id)
    {
         GameStateController gs = (GameStateController)gameStateControllers.
get(id);

        gs.unload();
    }

    public void setStartingGameStateController(int id)
    {
        currGameState = (GameStateController)gameStateControllers.get(id);
    }

    public void setNextGameStateController(int id)
    {
        nextGameState = (GameStateController)gameStateControllers.get(id);
    }

    public void requestShutdown()
    {
        engineRunning = false;
    }

    public void run()
    {
        if(currGameState == null)
```

```
        {
            return;
        }

        currGameState.start();

        long currentTime;

        while(engineRunning == true)
        {
            currentTime = System.currentTimeMillis();

            currGameState.step(currentTime - lastTime);

            currGameState.draw(canvas.getGameGraphics());

            canvas.swapBuffers();

            if(nextGameState != null)
            {
                currGameState.stop();
                nextGameState.start();

                currGameState = nextGameState;
                nextGameState = null;
            }
        }

        canvas.dispose();
    }
}
```

GameCanvas.java

```
/*
 * GameCanvas
 */

package javaPlay;

import java.awt.Container;
import java.awt.Graphics;
import java.awt.GraphicsConfiguration;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.Toolkit;
import java.awt.image.BufferStrategy;
import java.util.logging.Level;
```

```
import java.util.logging.Logger;
import javax.swing.JFrame;

/**
 * @author VisionLab/PUC-Rio
 */
public class GameCanvas extends JFrame
{
    private final int defaultScreenWidth = 800;
    private final int defaultScreenHeight = 600;
    private Graphics g;
    private BufferStrategy bf;
    private int renderScreenStartX;
    private int renderScreenStartY;

    public GameCanvas(GraphicsConfiguration gc)
    {
        super(gc);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(defaultScreenWidth, defaultScreenHeight);
        setVisible(true);

        createBufferStrategy(2);

        renderScreenStartX = this.getContentPane().getLocationOnScreen().x;
        renderScreenStartY = this.getContentPane().getLocationOnScreen().y;

        bf = getBufferStrategy();
    }

    public int getRenderScreenStartX()
    {
        return renderScreenStartX;
    }

    public int getRenderScreenStartY()
    {
        return renderScreenStartY;
    }

    public Graphics getGameGraphics()
    {
        g = bf.getDrawGraphics();
        return g;
    }
    public void swapBuffers()
    {
```

```
        bf.show();
        g.dispose();
        Toolkit.getDefaultToolkit().sync();
    }
}
```

GameStateController.java

```
/*
 * GameStateController
 */

package javaPlay;

import java.awt.*;

/**
 * @author VisionLab/PUC-Rio
 */
public interface GameStateController
{
    public void load();
    public void unload();
    public void start();
    public void step(long timeElapsed);
    public void draw(Graphics g);
    public void stop();
}
```

Keyboard.java

```
/*
 * Keyboard
 */

package javaPlay;

import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.util.ArrayList;
import java.util.Hashtable;

/**
 * @author VisionLab/PUC-Rio
 */
public class Keyboard implements KeyListener
{
    private Hashtable keysPressed;
```

```
    public static final int UP_KEY = 38;
    public static final int LEFT_KEY = 37;
    public static final int RIGHT_KEY = 39;
    public static final int DOWN_KEY = 40;
    public static final int ESCAPE_KEY = 27;
    public static final int SPACE_KEY = 32;
    public static final int ENTER_KEY = 10;

    public Keyboard()
    {
        keysPressed = new Hashtable();
    }

    public synchronized boolean keyDown(int keyCode)
    {
        return keysPressed.contains(keyCode);
    }

    public void keyTyped(KeyEvent e)
    {

    }

    public synchronized void keyPressed(KeyEvent e)
    {
        if(keysPressed.contains(e.getKeyCode()) == false)
        {
            keysPressed.put(e.getKeyCode(), e.getKeyCode());
        }
    }

    public synchronized void keyReleased(KeyEvent e)
    {
        keysPressed.remove(e.getKeyCode());
    }
}
```

Mouse.java

```
/*
 * Mouse
 */

package javaPlay;

import java.awt.Point;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
```

```
/**
 * @author VisionLab/PUC-Rio
 */
public class Mouse implements MouseMotionListener, MouseListener
{
    private Point mousePos;
    private boolean leftButtonPressed;
    private boolean middleButtonPressed;
    private boolean rightButtonPressed;

    public Mouse()
    {
        mousePos = new Point(0, 0);
        leftButtonPressed = false;
        middleButtonPressed = false;
        rightButtonPressed = false;
    }

    public Point getMousePos()
    {
        return mousePos;
    }

    public boolean isLeftButtonPressed()
    {
        return leftButtonPressed;
    }

    public boolean isMiddleButtonPressed()
    {
        return middleButtonPressed;
    }

    public boolean isRightButtonPressed()
    {
        return rightButtonPressed;
    }

    public void mouseClicked(MouseEvent e)
    {

    }

    public void mousePressed(MouseEvent e)
    {
        switch(e.getButton())
        {
            case MouseEvent.BUTTON1:
```

```
                leftButtonPressed = ((e.getModifiersEx() & MouseEvent.BUTTON1_
DOWN_MASK) != 0);
                break;
            case MouseEvent.BUTTON2:
                 middleButtonPressed = ((e.getModifiersEx() & MouseEvent.BUT-
TON2_DOWN_MASK) != 0);
                break;
            case MouseEvent.BUTTON3:
                  rightButtonPressed = ((e.getModifiersEx() & MouseEvent.BUT-
TON3_DOWN_MASK) != 0);
                break;
        }
    }

    public void mouseReleased(MouseEvent e)
    {
        switch(e.getButton())
        {
            case MouseEvent.BUTTON1:
                leftButtonPressed = ((e.getModifiersEx() & MouseEvent.
                .BUTTON1_DOWN_MASK) != 0);
                break;
            case MouseEvent.BUTTON2:
                middleButtonPressed = ((e.getModifiersEx() & MouseEvent.
                .BUTTON2_DOWN_MASK) != 0);
                break;
            case MouseEvent.BUTTON3:
                rightButtonPressed = ((e.getModifiersEx() & MouseEvent.
                .BUTTON3_DOWN_MASK) != 0);
                break;
        }
    }

    public void mouseEntered(MouseEvent e)
    {

    }

    public void mouseExited(MouseEvent e)
    {

    }

    public void mouseDragged(MouseEvent e)
    {

    }
```

```
    public void mouseMoved(MouseEvent e)
    {
        mousePos = e.getPoint();
    }

}
```

Scene.java

```
/*
 * Scene
 */

package javaPlay;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.Toolkit;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * @author VisionLab/PUC-Rio
 */
public class Scene
{
    private Image backDrop;
    private Image[] tiles;
    private ArrayList tileLayer;
    private ArrayList overlays;
    private int drawStartX = 0;
    private int drawStartY = 0;
    private final int MAX_SLEEP_COUNT = 30;

    public void loadFromFile(String sceneFile) throws InterruptedException,
    FileNotFoundException, IOException, Exception
    {
        tileLayer = new ArrayList();
        overlays = new ArrayList();
```

```
 BufferedReader input = new BufferedReader(new FileReader
(new File(sceneFile)));

 //first read the number of tile images
 String line = input.readLine();

 int numOfTileImages = Integer.parseInt(line, 10);

 tiles = new Image[numOfTileImages];

 int count;

 for(int i = 0 ; i < numOfTileImages ; i++)
 {
     //read each tile image name
     line = input.readLine();

     tiles[i] = Toolkit.getDefaultToolkit().getImage(line);

     count = 0;
     while(tiles[i].getWidth(null) == -1)
     {
         Thread.sleep(1);

         count++;

         if(count == MAX_SLEEP_COUNT)
         {
             throw new Exception("image could not be loaded");
         }
     }
 }

 //now read the tile set map until the final
 //character is found "%"
 String endTileSet = "%";

 line = input.readLine();

 while(line.equals(endTileSet) != true)
 {
     ArrayList tileLine = new ArrayList();

     String[] tileIndices = line.split(",");

     for(int i = 0 ; i < tileIndices.length ; i++)
     {
         tileLine.add(Integer.parseInt(tileIndices[i]));
     }
```

```
            tileLayer.add(tileLine);

            line = input.readLine();
        }

        //now read the backdrop file
        line = input.readLine();

        backDrop = Toolkit.getDefaultToolkit().getImage(line);

        count = 0;
        while(backDrop.getWidth(null) == -1)
        {
            Thread.sleep(1);

            count++;

            if(count == MAX_SLEEP_COUNT)
            {
                throw new Exception("image could not be loaded");
            }
        }
    }

    public void addOverlay(GameObject overlay)
    {
        overlays.add(overlay);
    }

    public void removeOverlay(GameObject overlay)
    {
        overlays.remove(overlay);
    }

    public void setDrawStartPos(int drawStartX, int drawStartY)
    {
        this.drawStartX = drawStartX;
        this.drawStartY = drawStartY;
    }

    public void draw(Graphics g)
    {
        //first clear the scene
        GameCanvas canvas = GameEngine.getInstance().getGameCanvas();

        g.setColor(Color.BLACK);

        g.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
```

```
        //first draw the backdrop
        int startDrawX = canvas.getRenderScreenStartX() - drawStartX;
        int startDrawY = canvas.getRenderScreenStartY() - drawStartY;

        g.drawImage(backDrop, startDrawX, startDrawY, null);

        //now draw the tile set
        int tileWidth = tiles[0].getWidth(null);
        int tileHeight = tiles[0].getHeight(null);

        int line = 0;
        int drawY = startDrawY;

        do
        {
            ArrayList tileLine = (ArrayList)tileLayer.get(line);

            int drawX = startDrawX;

            for(int c = 0 ; c < tileLine.size() ; c++, drawX += tileWidth)
            {
                int idx = (Integer)tileLine.get(c);

                if(idx == 0)
                {
                    continue;
                }

                g.drawImage(tiles[idx-1], drawX, drawY, null);
            }

            drawY += tileHeight;
            line++;

        }while(line < tileLayer.size());

        //finally draw the overlays
        for(int i = 0 ; i < overlays.size() ; i++)
        {
            GameObject element = (GameObject)overlays.get(i);

            element.draw(g);
        }
    }

    public Vector getTilesFromRect(Point min, Point max)
    {
        Vector v = new Vector();
```

```
        GameCanvas canvas = GameEngine.getInstance().getGameCanvas();

        int startDrawX = canvas.getRenderScreenStartX() - drawStartX;
        int startDrawY = canvas.getRenderScreenStartY() - drawStartY;

        int tileWidth = tiles[0].getWidth(null);
        int tileHeight = tiles[0].getHeight(null);

        int line = 0;
        int drawY = startDrawY;

        do
        {
            ArrayList tileLine = (ArrayList)tileLayer.get(line);

            int drawX = startDrawX;

            for(int c = 0 ; c < tileLine.size() ; c++, drawX += tileWidth)
            {
                TileInfo tile = new TileInfo();

                tile.id = (Integer)tileLine.get(c);
                tile.min.x = drawX - canvas.getRenderScreenStartX();
                tile.min.y = drawY - canvas.getRenderScreenStartY();
                tile.max.x = drawX - canvas.getRenderScreenStartX()
                + tileWidth - 1;
                tile.max.y = drawY - canvas.getRenderScreenStartY()
                + tileHeight - 1;

                if((min.x > tile.max.x) || (max.x < tile.min.x))
                {
                    continue;
                }
                if((min.y > tile.max.y) || (max.y < tile.min.y))
                {
                    continue;
                }

                v.add(tile);
            }

            drawY += tileHeight;
            line++;

        }while(line < tileLayer.size());

        return v;
    }
```

```
    public void step(int timeElapsed)
    {
        for(int i = 0 ; i < overlays.size() ; i++)
        {
            GameObject element = (GameObject)overlays.get(i);

            element.step(timeElapsed);
        }
    }
}
```

Sprite.java

```
/*
 * Sprite
 */

package javaPlay;

import java.awt.Graphics;
import java.awt.Image;
import java.awt.Toolkit;

/**
 * @author VisionLab/PUC-Rio
 */
public class Sprite
{
    private Image image;
    private int animFrameCount;
    private int currAnimFrame;
    private int animFrameWidth;
    private int animFrameHeight;
    private int MAX_COUNT = 50;

    public Sprite(String filename, int animFrameCount, int animFrameWidth,
            int animFrameHeight) throws Exception
    {
        image = Toolkit.getDefaultToolkit().getImage(filename);

        int count = 0;

        while(image.getWidth(null) == -1)
        {
            Thread.sleep(1);
            count++;
```

```
            if(count == MAX_COUNT)
            {
                throw new Exception("image could not be loaded");
            }
        }

        this.animFrameCount = animFrameCount;
        this.animFrameWidth = animFrameWidth;
        this.animFrameHeight = animFrameHeight;

        this.currAnimFrame = 0;
    }

    public void setCurrAnimFrame(int frame)
    {
        currAnimFrame = frame;
    }

    public void draw(Graphics g, int x, int y)
    {
        GameCanvas canvas = GameEngine.getInstance().getGameCanvas();

        int xpos = canvas.getRenderScreenStartX() + x;
        int ypos = canvas.getRenderScreenStartY() + y;

        g.drawImage(image, xpos, ypos, xpos + animFrameWidth, ypos + animFra-
meHeight,
                currAnimFrame * animFrameWidth, 0, (currAnimFrame + 1) * ani-
mFrameWidth, animFrameHeight, null);
    }

    private Sprite(Image image, int animFrameCount,
            int currAnimFrame, int animFrameWidth, int animFrameHeight)
    {
        this.image = image;
        this.animFrameCount = animFrameCount;
        this.currAnimFrame = currAnimFrame;
        this.animFrameWidth = animFrameWidth;
        this.animFrameHeight = animFrameHeight;
    }

    public Sprite clone()
    {
        return new Sprite(image, animFrameCount, currAnimFrame,
                animFrameWidth, animFrameHeight);
    }
}
```

TileInfo.java

```
/*
 * TileInfo
 */

package javaPlay;

import java.awt.Point;

/**
 * @author VisionLab/PUC-Rio
 */
public class TileInfo
{
    public int id;
    public Point min;
    public Point max;

    public TileInfo()
    {
       min = new Point();
       max = new Point();
    }
}
```

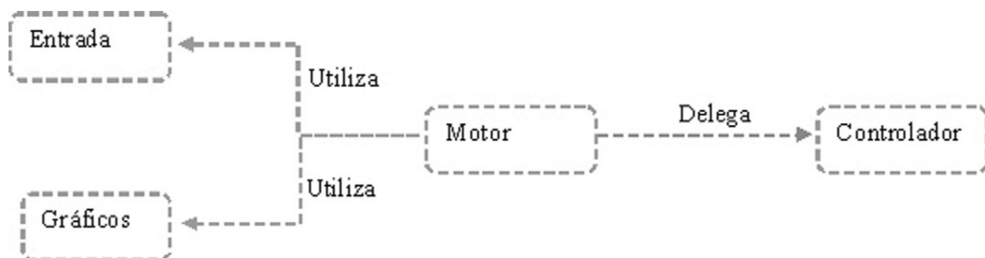
D

O Projeto do javaPlay

Este apêndice documenta os princípios de projeto do motor especialmente desenvolvido para este livro, o motor 2D javaPlay.

D.1. Arquitetura geral

O engine se divide em módulos responsáveis por tarefas específicas, chamadas **sistemas**. Dentre essas tarefas temos: controle de *loop* principal, desenho e verificação de *input* por parte dos jogadores etc. A figura a seguir mostra a interação desses sistemas.



Tais sistemas possuem diferentes representações e podem ser definidos por um conjunto de classes ou por uma classe apenas. Os sistemas são defini-

dos apenas para fornecer uma separação de tarefas e uma organização de alto nível para o *engine*.

O sistema de aplicação é o sistema central. Ele se utiliza dos outros sistemas e delega tarefas a outros capazes de realizar lógica específica de jogo.

D.2. Sistema Motor

O sistema Motor é responsável pelo controle do *loop* principal de um jogo. Ele acessa outros sistemas que realizam tarefas específicas. O sistema de aplicação é responsável pela inicialização de um programa e dos outros sistemas, bem como do término do programa e finalização correta dos outros sistemas.

Ele define um conjunto de funções predefinidas que permitem ao programador de aplicação controle sobre a inicialização, o término e as ações ocorridas para cada *frame* de jogo.

O sistema de aplicação possui uma classe principal *singleton* que permite que as diferentes partes do sistema possuam visibilidade das classes mais importantes como o sistema gráfico e o sistema de entrada e saída. Essa classe se chama **GameEngine**. **GameEngine** possui os seguintes métodos:

```
public static getInstance()
public Keyboard getKeyboard()
public Mouse getMouse()
public GameCanvas getGameCanvas()
public void addGameStateController(StateController,int controllerId)
public void removeGameStateController(int controllerId)
public void setNextGameStateController(int controllerId)
```

O método estático getInstance retorna a instância de *singleton* da aplicação. Dessa maneira, o acesso desse objeto, em basicamente qualquer ponto da aplicação, se torna trivial. getKeyboard e getMouse retornam os objetos de controle de entrada de usuário. getGraphicsManager retorna o objeto GraphicsManager corrente, permitindo acesso a, por exemplo, o contexto de renderização corrente.

A inicialização e as funções executadas ao término da aplicação são associadas a uma interface denominada GameStateController.

D.3. Sistema de estado

O sistema de estado é o principal responsável pela flexibilidade do sistema. Enquanto a classe de aplicação encapsula as funções de baixo nível associadas à plataforma e aos métodos globais de controle, o sistema de estado permite que o programador tenha um controle fino das ações que deverão ser apresentadas ao usuário final.

Jogos apresentam diferentes requisitos lógicos de acordo com o estado em que se encontram. Por exemplo, a lógica necessária para tratar o menu principal de jogo difere grandemente da lógica de jogo principal, em que o usuário de fato controla seu avatar. Para o sistema de aplicação, ambos os estados são idênticos, uma vez que este tem como função apenas preparar o ambiente para a execução coerente de um novo *frame*. A lógica que será processada nesse *frame* é baseada em um estado derivado da classe **GameState**.

A classe GameStateController é abstrata e permite a execução de lógica arbitrária. Como o sistema de aplicação, ela divide a execução em *time slices*, permitindo ao programador o controle de eventos específicos como processamento de *frame*. Os principais métodos dessa classe são:

```
public boolean load()
public boolean unload()
public boolean start()
public void step(int timeElapsed)
public void draw()
public boolean stop()
```

Os métodos load e unload são invocados quando o estado deve ser carregado ou descarregado da memória. Todos os recursos devem passar pelo mesmo processo.

Os métodos start e stop são invocados quando o controlador de estado atual deve ser modificado, mas seus recursos não devem ser liberados. Esse cenário é comum quando se deseja que uma lógica específica seja executada, mas a troca deve ser rápida, não existindo a possibilidade de pausa para carregamento de recursos.

Os métodos step e draw permitem o controle fino da lógica associada a um *frame* específico. O sistema de aplicação invoca o método step passando o tempo em milissegundos decorrente entre o *frame* atual e o *frame* anterior,

permitindo assim animações baseadas em tempo. Nesse método o programador deve atualizar o estado de todos os objetos de jogo relevantes e prepará-los para a renderização.

Após a invocação do método step, o sistema de aplicação chama o método draw do controlador atual. Este, por sua vez, permite ao programador renderizar seus objetos de forma customizada, ou repassar a uma entidade de alto nível como a classe Scene.

D.4. Objetos de jogo

O engine provê também uma classe base utilizada para qualquer objeto que necessite de lógica específica de atualização.

GameObject é uma classe abstrata que define um objeto com posição no espaço e a capacidade de receber mensagens de atualização e realizar operações de desenho customizadas. A classe GameObject é definida por:

```
public int x,y;
public void step(int timeElapsed)
public void draw()
```

É fácil perceber que essa classe por si só não apresenta muitas funcionalidades. Ela serve apenas como um ponto de partida. É de responsabilidade das classes derivadas adicionar atributos, como um objeto sprite que representa o gráfico do avatar e métodos específicos.

D.5. Sistemas de entrada

O sistema de entrada permite acesso ao estado do teclado e mouse para uma determinada aplicação. As principais classes desse sistema são Keyboard e Mouse. Os métodos públicos da classe Keyboard são:

```
public boolean keyPressed(byte keyCode)
public boolean keyReleased(byte keyCode)
public boolean keyDown(byte keyCode)
```

O método keyPressed permite verificar se uma tecla foi pressionada no *frame* corrente de execuação. keyReleased verifica se a tecla deixou de ser pressionada no *frame* corrente. keyDown verifica o estado da tecla independente do último *frame* em que foi modificado.

Os métodos da classe Mouse são:

```
public Point mousePos()
public boolean isLeftButtonPressed()
public boolean isMiddleButtonPressed()
public boolean isRightButtonPressed()
public Point getMousePos()
```

O método getMousePos retorna um objeto do tipo Point com as coordenadas x e y da última localização do mouse antes de entrar no *frame* corrente. isLeftButtonPressed verifica se o botão esquerdo do mouse se encontra pressionado. isRightButtonPressed faz o mesmo para o botão direito.

D.6. Sistema gráfico

O sistema gráfico deve se dividir em duas camadas básicas: Graphics Manager e a Scene Manager. A camada Graphics Manager é responsável pela interação do *engine* com as camadas de desenho da plataforma.

A camada GraphicsManager é definida para ser o mais leve possível, sendo capaz de permitir a integração dos métodos de desenho em um método controlado pelo sistema de aplicação em vez de eventos assíncronos enviados pelo *runtime* Java.

Ela fornece acesso aos objetos AWT, permitindo que o usuário tenha acesso ao contexto gráfico corrente e possa utilizar os métodos de desenho encontrados nas classes Graphics e Graphics2D.

A classe Scene define uma abstração de alto nível para criação de jogos baseados em mundos formados por blocos conhecidos como **Tiles**. Esse modelo de organização de mundo é o mais comum encontrado em jogos 2D. Essa classe divide o mundo em subcamadas distintas:

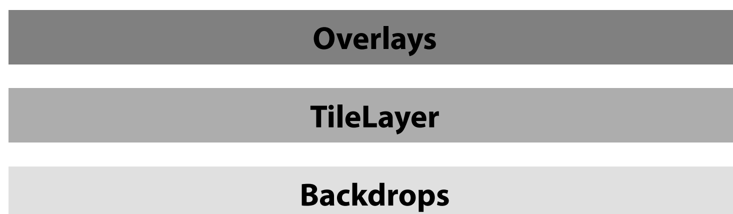
- Backdrops
- TileLayer
- Overlays

Backdrops são os cenários de fundo, sobre os quais o usuário possui pouca ou nenhuma interação, servindo na maioria das vezes como itens de adorno gráfico.

TileLayer é a subcamada que apresenta o conjunto de blocos que define o mundo ao qual um avatar é inserido. Ela fornece métodos para detecção de colisão com os blocos do mundo.

A subcamada de Overlays representa o local onde os sprites dos objetos dinâmicos de jogo, controlados pelo jogador, ou inteligência artificial, são desenhados. Esses objetos são representados por sprites. Um overlay pode ser entendido como uma camada inicialmente transparente na qual são desenhados os sprites do avatar, inimigos etc.

A forma como Scene organiza suas subcamadas é representada na seguinte figura:



A classe é capaz de realizar operações de *blending* das camadas, e permite a renderização em ordem correta de profundidade dos objetos: objetos de overlay se situam à frente dos objetos de tile, que, por sua vez, estão à frente dos backdrops.

Os métodos encontrados nessa classe são:

```
public static Scene loadFromFile(string sceneFile)
public void addOverlay(GameObject overlay)
public void removeOverlay(GameObject overlay)

public Vector getTilesFromRect(Point min, Point max)

public void draw()
public void step(timeElapsed)
```

Os métodos addOverlay, overlays e removeOverlay permitem a inserção, deleção e busca de objetos de overlay respectivamente. Métodos análogos são disponíveis para backdrops.

O método step(int timeElapsed) é invocado pelo StateController e invoca step de cada um dos objetos de jogo associados ao SceneManager. O método draw, por sua vez, desenha os backdrops, conjunto de tiles e conjunto de overlays.

O método getTilesFromRect recebe duas posições representando um retângulo e retorna os tiles ocupados por este. Com isso é possível saber quais são os tiles que um determinado objeto ocupa e, através dessa informação, decidir se há colisão, se é necessário algum tipo de processamento especial etc.

D.7. Sprites

A classe Sprite representa os principais objetos gráficos a serem manipulados em jogos 2D. Os sprites representam em geral todas as poses que um objeto pode ficar em suas animações. Sem sprites animados não seria possível realizar as complexas animações de personagens vistas em jogos. A classe sprite possui a seguinte interface:

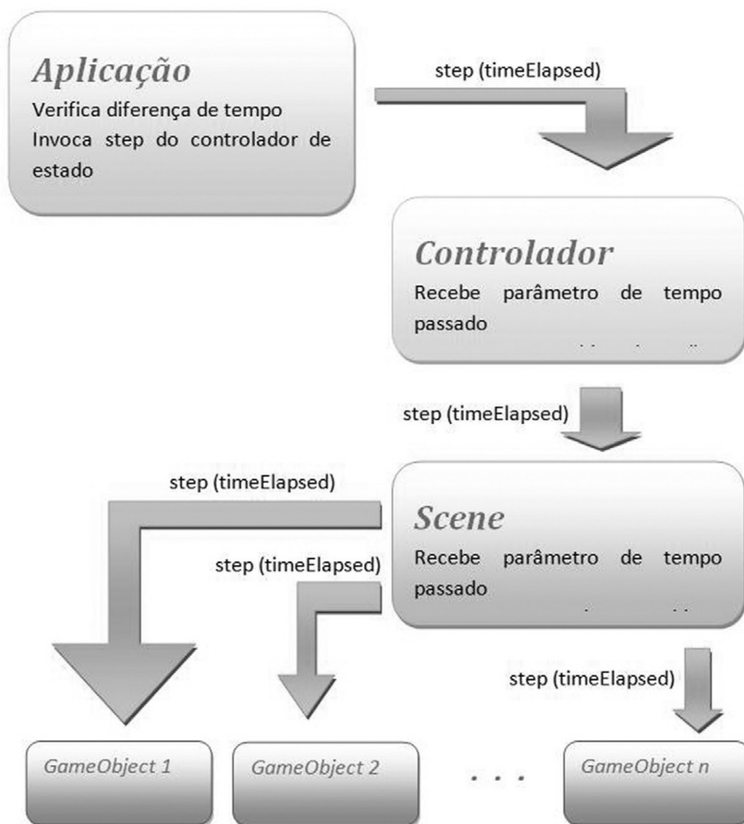
```
public x,y;
public int setCurrAnimFrame(int frame)
public void draw(Graphics g)
public Sprite clone()
```

Os atributos x e y definem a posição do sprite no canvas de desenho. getCurrentAnimFrame e setCurrentAnimFrame definem o *frame* atual de animação. getAnimFrameRect retorna o retângulo ao qual o *frame* corrente pertence. Por exemplo, se cada *frame* de animação possui altura de 90 pixels e largura de 20 pixels, os extremos do retângulo associado ao *frame* 2 (assumindo que a contagem de *frames* começa em 0) é [(40,0),(60,90)].

Como comodidade, é fornecido um método draw que desenha o *frame* atual de animação com origem no ponto (x,y) no contexto de renderização associado ao parâmetro g.

D.8. Exemplo de loop

O *loop* principal do *engine* se inicia dentro da classe MainApplication que possui controle da *thread* principal de jogo. Ela verifica a diferença de tempo entre o *frame* anterior e o *frame* atual em milissegundos, e passa a execução para o método step do controlador de estado corrente. O controlador então pode executar sua lógica de atualização. Se o controlador tiver um objeto de alto nível associado (como, por exemplo, uma instância da classe Scene), deve invocar o método step de cena, que, por sua vez, atualizará cada um de seus GameObjetcs.



É importante lembrar que o modelo aqui utilizado é semelhante ao paradigma de *time sharing*: cada objeto recebe um "quantum" de tempo e este deve utilizar esse parâmetro para atualizar seu estado corretamente. Esse procedimento é repetido para cada objeto até que todos tenham sido atualizados. Tal lógica difere de atualizações feitas através de iterações tradicionais em laço do tipo *while* ou *for*, em que um objeto realizaria toda sua atualização sequencialmente.

Índice Remissivo